COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-4055

# ANNA PACKAGE SPECIFICATION: CASE STUDIES

**Edited by John J. Kenney and Walter Mann**
**Contributions from members of the Program Analysis and Verification Group.**

**Technical Report: CSL-TR-91-496**

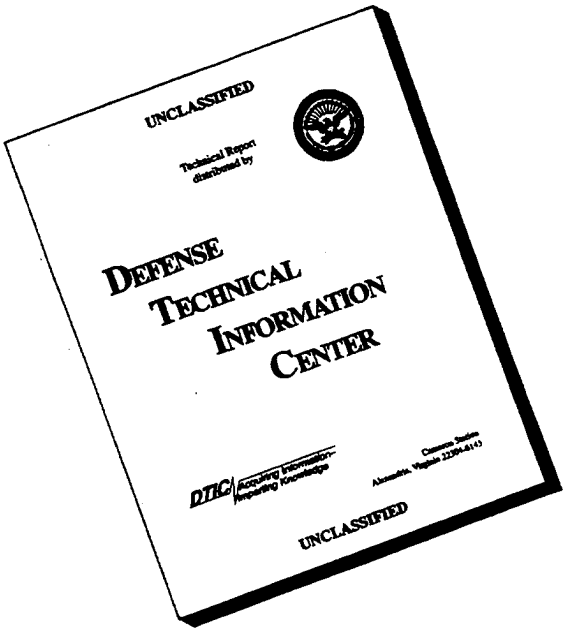
(Program Analysis and Verification Group Report No. 56)



**October 1991**

This research has been supported by the Advanced Research Projects Agency, Department of Defense, under contracts N00039-82-C-0250, N00039-84-C-0211 and N00039-91-C-0162.

# DISCLAIMER NOTICE




**THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.**

# Anna Package Specification: Case Studies

Edited by John J. Kenney and Walter Mann
Contributions from members of the Program Analysis and Verification Group.

**Technical Report: CSL-TR-91-496**
Program Analysis and Verification Group Report No. 56

October 1991

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

**Abstract**

We present specification techniques of Ada[1] software, based on the Anna specification language, and examples of Ada packages formally specified in Anna. A package specification for an abstract set data type is used to illustrate the techniques and pitfalls involved in the process of software specification and development. This specification not only exemplifies good Anna style and specification approach, but has a secondary goal of teaching the reader how to use Anna and the associated set of Anna tools developed at Stanford University over the past six years. Additional packages are presented which reflect a variety of styles and approaches to specify Ada packages. The technical report thus aims to give readers a new way of looking at the software design and development process, synthesizing fifteen years of research in the process.

**Acknowledgements**

The development of the examples in this technical report has taken place over the past five years. Earlier versions of this set of examples were edited by Will Tracz and Randall Neff.

Examples have been contributed by various members of the Program Analysis and Verification Group at Stanford. The contributors include Doug Bryan, Rob Chang, John Kenney, Chuan-Chieh Ko, David Luckham, Neel Madhav, Walter Mann, Geoff Mendal, Randall Neff, David Rosenblum, Sriram Sankar and Will Tracz.

**Key Words and Phrases:** abstract data type, Ada, Anna, formal methods, formal specification.

[1] Ada is a registed trademark of the U.S. Government (Ada Joint Program Office).

Program Analysis and Verification Group
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

# Contents

# Preface

Anna is a language extension of Ada to include facilities for formally specifying the intended behavior of Ada programs. It is designed to meet a perceived need to augment Ada with precise machine-processable annotations so that well established formal methods of specification and documentation can be applied to Ada programs. The annotations are well suited for different possible applications during the life cycle of a program. Such applications include not only testing, debugging and formal verification of a finished program, but also specification of program parts during the earlier stages of requirements analysis and program design.

This technical report is designed to present specification techniques of Ada software, based on the Anna specification language. We do this by presenting examples of Ada packages formally specified in Anna. These examples were developed by persons with different backgrounds and levels of training. The examples, therefore, reflect a variety of styles and approaches to specifying Ada packages. A package specification for an abstract set data type will be delved into in detail to illustrate the techniques and pitfalls involved in the process of software specification and development. This specification not only exemplifies good Anna style and specification approach, but has a secondary goal of teaching the reader how to use Anna and the associated set of Anna tools developed at Stanford University over the past six years. The technical report thus aims to give readers a new way of looking at the software design and development process, synthesizing fifteen years of research in the process.

One motivation for this report is that there has been growing interest in Anna, and we felt it was time to produce a library of Anna package specifications, which can be used to exemplify and teach the practical application of Anna to real Ada software packages. It is therefore a pleasure to introduce this collection of examples. We hope that you will experiment with these methodologies and specifications, and use them in your Ada software development strategy. This technical report is designed to complement both the Anna reference manual [6], and "Programming With Specifications: An Introduction to Anna" by David Luckham [5].

**Acknowledgements**

The development of the examples in this technical report has taken place over the past five years. Earlier versions of this set of examples were edited by Will Tracz and Randall Neff.

Examples have been contributed by various members of the Program Analysis and Verification Group at Stanford. The contributors include Doug Bryan, Rob Chang, John Kenney, Chuan-Chieh Ko, David Luckham, Neel Madhav, Walter Mann, Geoff Mendal, Randall Neff, David Rosenblum, Sriram Sankar and Will Tracz.

# Chapter 1

# Introduction

*Anna* (**ANN**otated **A**da) [4,5,6] is a language extension of Ada[19] to include facilities for formally specifying the intended behavior of Ada programs. It is designed to meet a perceived need to augment Ada with precise machine-processable annotations so that well established formal methods of specification and documentation can be applied to Ada programs. Essentially, this language provide facilities for expressing information about programs that is not normally part of the program text. The specifications are expressed in a machine processable form — they can be parsed, checked for static semantic errors, and in many cases, compiled into run-time tests. The specifications are well suited for different possible applications during the life cycle of a program. Such applications include not only testing, debugging and formal verification of a finished program, but also specification of program parts during the earlier stages of requirements analysis and program design.

## 1.1 Formal Methods

Formal methods used in developing computer systems are mathematically based techniques for describing system properties. They provide frameworks for specifying, developing, and verifying systems in a systematic, rather than ad hoc, manner. System designers use formal methods to specify a system's desired behavioral and structural properties. Formal methods can be used in all phases of a system's development, and present an opportunity to develop new techniques to improve software production.

One tangible product of applying a formal method is a formal specification. A specification serves as a contract, a valuable piece of documentation, and a means of communication among a client, a specifier, and an implementor. Formal specifications have the additional advantage over informal ones of being amenable to machine analysis and manipulation. The greatest benefit of applying a formal method is an intangible one, which often comes from the process of formalizing rather than from the specification. System designers will gain a deeper understanding of the specified system because they have been forced to be more abstract and precise about the desired properties. But it is not necessary to prove every property or detail about a system; proving small things about core properties of the system will be invaluable in the future.

Consider for each system development phase, some uses of formal specifications and the formal methods that support them.

**Requirements analysis.** This step clarifies the informally stated requirements, helps clear up vague ideas, reveals contradictions (or inconsistencies), ambiguities and incompleteness.

**System design.** This step assists during decomposition and refinement by recording design decisions and assumptions. An interface specification provides its clients the information needed to use the module without knowledge of its implementation. At the same time, it provides the implementor the information needed to construct the modules without knowledge of its clients. The interface may remain the

same, and the implementation can be replaced, perhaps by a more efficient one, without hurting the client.

**System verification.** Verification is the process of showing that a system satisfies its specification. Verification is impossible without a formal specification. It is important to realize that although the entire system may never be completely verified, a smaller, critical piece often can be.

**System validation.** Formal methods can aid in system testing and debugging. Specifications can be used to generate complete test cases.

**System documentation.** A specification serves as a description of the system. It communicates between a client and a specifier, between a specifier and an implementor, and among the implementation team. Formal methods are intended to capture the "what" rather than the "how."

**System analysis and evaluation.** To learn from the experience of building a prototype system, developers should do a critical analysis of its functionality and performance after it has been built. In fact, much recent work has been done in specifying a system which is already built, running, and used. Some of these exercises revealed serious bugs in published algorithms and circuit designs. As to be expected, most revealed unstated assumptions, inconsistencies, and unintentional incompleteness in the system.

## 1.2 Package Specification

This technical report is designed to present specification techniques of Ada software, based on the Anna specification language. We do this by presenting examples of Ada packages formally specified in Anna. The examples are arranged in increasing order of complexity. These examples were developed by persons with different backgrounds and levels of training. This in itself we hope will be useful to the reader. The examples, therefore, reflect a variety of styles and approaches to specifying Ada packages. The examples demonstrate various annotation methodologies and features of Anna. This collection can be used both as a reference guide for the experienced Anna programmer and as a tutorial for the novice specification programmer.

A package specification for an abstract set data type will be delved into in detail to illustrate the techniques and pitfalls involved in the process of software specification and development. This specification not only exemplifies good Anna style and specification approach, but has a secondary goal of teaching the reader how to use Anna and the associated set of Anna tools developed at Stanford University over the past six years. The Anna toolset includes a semanticizer, specification analyzer, and an Anna to Ada transformer, which compiles the annotations into run-time tests. The technical report thus aims to give readers a new way of looking at the software design and development process, synthesizing fifteen years of research in the process.

Possible uses of Anna are:

**Formal Standards.** The Anna package specifications provide precise, detailed, machine independent definition of a package. Anna can be used as a formal definition for a package.

**Testing and Debugging.** Recognition of inconsistencies between Anna specifications and Ada programs is automated. The Anna specifications are automatically converted into Ada checking code [14,16] by a tool called the run-time checking transformer. The transformation therefore produces a self-checking program which can interact with other tools (e.g. a symbolic debugger). When the transformed program is executed on test cases, inconsistencies are automatically detected and diagnosed.

**Rapid Prototyping.** Anna can be used to provide a quick set of specifications for a particular problem. These specifications would be at a higher level and simpler and less detailed than the actual Ada code that will eventually be written. Techniques of symbolic execution and inferential analysis can be applied to specification, to provide most of the results sought by rapid prototypes. The Specification

Analyzer [2,9,11] is a tool designed for this use of Anna. Indeed, in the future, these symbolic reasoning techniques can also be combined with partial implementations to provide more detailed prototyping.

**Top Down Formal Design.** Informal requirements for a package are converted into Anna specifications. Symbolic execution permits the testing and debugging of the specifications. The Specification Analyzer provides support for various activities, such as development and analysis, in the transition between informal requirements and accurate formal specifications. Formal specifications can then be used as a guide for the manual or automatic construction of an implementation. They can also be used to guide the decomposition of the formal specification into several component packages.

Specifying Ada packages using Anna involves several techniques. One of the more important techniques is *abstraction* or *information hiding*. Anna is used to express the behavior semantics of an Ada package in an abstract manner. Writing Anna specifications can be thought of as a new form of programming. Different kinds of decisions have to be made as compared to those made during traditional programming. Some of the decisions involved are in the design of packages — the kind of data types and operations that comprise the package; and the design of specifications that promise enough about the package to make the package useful; while at the same time not over-specifying the package, thus adversely constraining its implementation. Other decisions to be made are the kinds of specification techniques to use — the package may be specified either operationally or axiomatically; or the specification of a package may make use of concepts defined in other packages. This process of re-using concepts is termed *relative specification*. These specification processes are techniques of abstraction, and as yet, are in their infancy. Our examples are intended to illustrate the choices involved in abstractions, and some simple techniques of abstraction.

However, the measures by which a program is judged are somewhat different from the measures for an abstraction. Programs are concerned with correct behavior, economy of space, and speed. Understandability has traditionally been a secondary consideration. Abstractions are much more concerned with understandability, because their purpose is to provide guidelines and standards. The first measure is consistency, then consequences and implications. Elegance, brevity, simplicity, delayed commitment — these are some of the measures of abstractions. In this report, the reader is exposed to various kinds of annotations for abstract specifications of Ada packages, and their use or misuse.

## 1.3 Overview

The structure of the technical report is as follows: Chapter 2 gives an overview of Anna. The reader should be able to understand and construct straightforward Anna specifications after reading this chapter. In Chapter 3, we design and develop a Set package. Various design issues are addressed and different solutions are proposed. The remaining chapters describe additional Ada packages formally specified in Anna. Chapter 4 contains a number of abstract data types. We continue in Chapter 5 with some simple package specifications that are not definitions of abstract data types. Finally, Chapter 6 presents four complex examples that show the use of Anna on large, real world packages.

This set of Anna package specifications has been collected into a library which can be used to exemplify and teach the practical application of Anna to real Ada software packages. This library of specifications as well as some implementations for them are available via anonymous ftp. We hope that you will experiment with these methodologies and specifications, and use them in your Ada software development strategy. Contact the Anna team at the address below for more details.

For more information about the Anna language and the Anna-I toolset, contact the following address:

Stanford Anna Team
Computer Systems Lab, ERL 456
Stanford University
Stanford, CA 94305

(415) 723-1175 (voice)
(415) 725-7398 (fax)
anna-request@anna.stanford.edu

# Chapter 2

# Overview of Anna

*Anna* (**ANN**otated **A**da) [4,5,6] is a language extension of Ada to include facilities for formally specifying the intended behavior of Ada programs. The design of Anna was initiated in 1980 by Bernd Krieg-Brückner and David Luckham. They were joined by Olaf Owe and Friedrich W. von Henke during the subsequent development stages of the Anna design. The current Anna design is based on the ANSI standard version of Ada and includes annotations for all Ada constructs except tasking.

Anna is based on first-order logic and its syntax is a straightforward extension of the Ada syntax. Most new concepts in Anna are extensions of concepts already in Ada. For example, concepts such as scope, visibility and overload resolution also apply to Anna constructs. Anna constructs appear as *formal comments* within the Ada source text (within the Ada comment framework). Therefore, from the point of view of Ada, formal comments are just comments and hence *Anna programs* (Ada programs with Anna specifications) can be accepted by Ada compilers and other Ada tools. [2]

## 2.1 Anna Formal Comments

Anna defines two kinds of formal comments, which are introduced by special comment indicators in order to distinguish them from informal comments. These formal comments are *virtual Ada text*, each line of which begins with the indicator `--:`, and *annotations*, each line of which begins with the indicator `--|`.

### 2.1.1 Virtual Ada Text

The purpose of virtual Ada text is to define *concepts* used in annotations. Often the formal specifications of a program will refer to concepts that are not explicitly implemented as part of the program. These concepts can be defined as virtual Ada text declarations. Virtual Ada text may also be used to *compute* values that are not computed by the actual program, but that are useful in defining the behavior of the program.

Virtual Ada text is Ada text with a few minor exceptions. That is, if all the virtual Ada text formal comment indicators (`--:`) are deleted from an Anna program, then the resulting program is a legal Ada program (with the few minor exceptions referred to earlier). However, the virtual Ada text must be such that it does not modify the semantics of the *underlying* Ada program. To achieve this the following restrictions are placed on virtual Ada text:

- Virtual Ada text may not syntactically contain actual Ada text. For example, it is not permitted to enclose actual Ada statements in a virtual Ada block.
- Execution of virtual Ada text statements may not change (directly or indirectly) the values of actual Ada objects.

[2] This chapter is reproduced with permission from [14] copyright 1990.

- Virtual Ada text declarations may not hide any actual Ada declarations.
- Execution of virtual Ada text statements may not change the flow of control of the underlying program. Thus *return*, *exit* and *goto* statements within virtual Ada text can transfer control only within the largest enclosing virtual block or body.

A STACK package is shown below augmented with a virtual concept—LENGTH. This virtual concept can be used to specify the operations PUSH and POP as will be described later.

```
   package STACK is
--:   function LENGTH return INTEGER;
      procedure PUSH ( E : in ELEMENT );
      function POP return ELEMENT;
   end STACK;
```

### 2.1.2 Annotations

Annotations are constraints on the underlying Ada program. They are made up of expressions which are typically boolean-valued. The location of the annotation in the Ada program together with its syntactic structure indicates the kind of constraints that the annotation imposes on the underlying program. Anna provides different kinds of annotations, each associated with a particular Ada construct. These are annotations of objects, types and subtypes, statements, and subprograms; in addition there are *axiomatic* annotations of packages, *propagation* annotations of exceptions and *context* annotations of entity visibility.

In addition to Ada expressions, expressions in annotations can also contain *quantified expressions*, *conditional expressions*, *state expressions* and *Anna membership tests*. Every annotation has a region of Anna text over which it applies, called its *scope*. The scope of an annotation is determined by the Ada scoping and visibility rules based on the position of the annotation in the Anna program. For example, if the annotation occurs in the position of a declaration, its scope extends from its position to the end of that declarative region. Generally, annotations constrain all *observable states* within their scope. An observable state is one that results from either the elaboration of a declaration or by the execution of a simple statement. What this means is that annotations do not constrain intermediate program states that occur during the execution of simple statements.

## 2.2 Anna Execution Semantics

When an Anna program is executed, each observable state encountered has to satisfy all Anna constraints whose scope includes this state. In case the program does not satisfy any one of these Anna constraints, the predefined Anna exception ANNA_ERROR is raised at this point. This exception can be handled using a virtual exception handler for ANNA_ERROR or an exception handler with the *others* option. This facility allows for some amount of error recovery.

## 2.3 Anna Expressions

### 2.3.1 Quantified Expressions

Both universal and existential quantifiers can be used in Anna expressions. The quantified variables are referred to as *logical variables*. Logical variables can be quantified over a range of values or over all the values of a type. The range of quantification is also further constrained to only those values for which the quantified expression is defined. A few examples are shown below:

```
for all X, Y : NATURAL => X - (( X / Y ) * Y ) = X mod Y
-- Note that the range of quantification for Y does not include 0.

exist X : INTEGER => X * X = 100
```

### 2.3.2 Conditional Expressions

*Conditional expressions* are expressions which can take on one of a set of values depending on the values of a set of guards. An example is shown below:

```
F( X ) = if X = 0 then
             1
         elsif X = 1 then
             1
         else
             F( X - 1 ) + F( X - 2 )
         end if
```

The guards are evaluated in sequence from the beginning until one of the guards evaluates to TRUE. The value of the expression corresponding to this guard then becomes the value of the conditional expression. If all guards evaluate to FALSE then the value of the expression in the *else part* becomes the value of the conditional expression. All conditional expressions must contain an *else part.*

### 2.3.3 State Expressions

*State expressions* are useful to describe values of composite objects, especially as a result of a modification to one of their components. State expressions of arrays, records and package states are described in the following paragraphs. A fourth form — state expressions of collections of access types — will not be dealt with here. State expressions have the following format:

*value* [ *modification* ]

Here, *value* is of the composite type and *modification* describes a modification to one of its components. A short form is available to describe values that result from sequences of such modifications:

*value* [ $modification_1$ ; $modification_2$ ]

is equivalent to:

*value* [ $modification_1$ ] [ $modification_2$ ]

#### Array and Record State Expressions

The *modification* of array and record state expressions have the following format:

*component* => *expression*

Examples are shown below:

```
STR[3 => 'I']
-- This expression has the value of array STR except that the third component is replaced by 'I'.
JOHN[AGE => 36; WIFE_NAME => "LINDA␣␣␣␣␣"]
-- This expression has the value of record JOHN except that the components AGE and WIFE_NAME are
-- modified.
```

**Package States and Package State Expressions**

A *package state* in Anna is a value that represents the state of a package. The package state is modeled as a record whose components are the variables declared immediately within the body of the package. There are other components that form part of the package state such as the states of nested packages, but they will not be dealt with here. Within the package body, where the details of the package state are visible, package state values can be used in about the same way as record values. Outside the package body, package state values behave similarly to private type values. Anna defines two attributes of packages that are used in conjunction with package states. If P is a package, P'STATE denotes the *current state* of the package P. For notational convenience in annotations, the current state may be denoted by the package name itself; *i.e.*, the attribute designator STATE may be omitted. P'TYPE denotes the implicit record type that models the package state. *Package state expressions* describe the effects on the package state as a result of executing package operations.

The *modification* of package state expressions takes the form of a call to a package subprogram. The value of the expression is the value of the package state that results when the subprogram call is executed in the specified package state. Some examples are shown below:

```
STACK'STATE[PUSH(E1)]
-- The state of the package STACK as a result of executing PUSH(E1) on the current state of this package.
STACK[PUSH(E2); POP]
-- Here, 'STATE has been omitted for notational convenience. This expressions denotes the state of the
-- package STACK as a result of executing PUSH(E2) followed by POP on the current state of this package
```

There is another useful operation on package states. The expression S.F(...) where S is a package state expression and F is a package function denotes the value F(...) returns if it is called when the package is in state S. An example is shown below:

```
STACK[PUSH(E2)].POP
```

### 2.3.4 Initial Expressions

An *initial expression* contains the keyword **in** (referred to as the modifier) followed by an expression referred to as the *modified expression*. Initial expressions are constants. The value of an initial expression is the value that the corresponding modified expression had when it was elaborated. During elaboration the initial expression is replaced by the value of the expression it modifies. Some examples are shown below:

```
in X
in (X**2 + Y**2)
```

### 2.3.5 Anna Operators

There are four new operators in Anna. They are the implication operator, the equivalence operator, Anna relational operators and Anna membership tests. The implication operator (–>) and the equivalence operator (<–>) have their usual meanings:

```
A –> B   -- if A then B
A <–> B  -- A if and only if B
```

The Anna relational operators feature a syntactic short-hand to express certain commonly occurring conjunctions of Ada relations. This is best described by an example:

```
A < B <= C
```

is equivalent to:

```
A < B and B <= C
```

The Anna membership test (**isin**) is an extension of the Ada membership test. The Ada membership test (A **in** T) checks that the value A satisfies the Ada constraints on T. The Anna membership test (A **isin** T) checks that the value A satisfies both the Ada and the Anna constraints on T. Anna constraints can be placed on T using subtype annotations (described later).

## 2.4 Annotations

### 2.4.1 Object Annotations

An object annotation is a BOOLEAN expression that constrains the values of the variables occurring in this expression throughout the scope of the annotation. Object annotations can occur in declarative regions only. In the special case in which there are no variables in the expression, the annotation is a constant and is therefore a constraint at the point of elaboration of the annotation. An example of this special case is an object annotation whose expression is an initial expression. There is another kind of object annotation, the *out annotation* which constrains only the points of exit from the scope. Some examples of object annotations are shown below:

```
--| CIRCUMFERENCE = 3.14159 * DIAMETER;
-- This object annotation constrains the variables CIRCUMFERENCE and DIAMETER throughout the
-- scope of the annotation.

--| in( X > 0 );
-- An object annotation with no variables (though X is a variable, it is within an initial expression). This
-- annotation constrains the point at which the annotation is defined.

--| out( Y = in X );
-- An out annotation which constrains the variable Y (on exit from the scope of the annotation) to be equal
-- to the value X had when the annotation was elaborated.
```

### 2.4.2 Subtype Annotations

A subtype annotation is a constraint on types and subtypes. Unlike object annotations, there can be only one subtype annotation for each type or subtype definition. An example of a subtype annotation is shown below:

```
    type EVEN is new INTEGER;
--| where X : EVEN => X mod 2 = 0;
```

This annotation constrains all objects X of the type EVEN to satisfy the constraint X **mod** 2 = 0. If the subtype annotation contains any variables other than the logical variable (X in the above example), then these variables are implicitly modified by **in** (they are replaced by their values at elaboration time). Hence subtype annotations only constrain the corresponding types or subtypes.

### 2.4.3 Statement Annotations

There are two different kinds of statement annotations—*simple statement annotations* and *compound statement annotations.*

#### Simple Statement Annotations

Simple statement annotations are constraints on a single statement. This constrained statement is the one immediately before the simple statement annotation. The constraint imposed by the annotation has to hold when control leaves the constrained statement—*i.e.*, it behaves like an *out annotation* on the constrained statement. If the annotation occurs at the beginning of a sequence of statements, then it constrains an implicit *null statement* just before the annotation.

In most cases, the simple statement annotation is a constraint that has to hold whenever control passes the point where the annotation is located. However, when the preceding statement transfers control to some other location (as is the case with *goto*, *return* and *exit* statements) then the constraint has to hold just before control is transferred by this statement. An example is given below:

```
    I := 2;
    while ( I <= N ) loop
        if A( I - 1 ) > A( I ) then
            EXCHANGE( A( I - 1 ), A( I ));
        end if;
--|     A( I ) >= A( I - 1 );
        -- This constraint has to hold after the execution of the preceding if statement.
        I := I + 1;
    end loop;
```

#### Compound Statement Annotations

Compound statement annotations are constraints on compound statements. The constrained statement occurs immediately after the compound statement annotation. The annotation is bound to the statement by the keyword **with** and constrains all observable states in the compound statement—*i.e.*, it behaves like an object annotation on the constrained statement. The previous example is shown below once again with a compound statement annotation included:

```
    I := 2;
--| with 1 < I <= N + 1;
    -- This constraint must hold at all observable states within the following while loop.
    while I <= N loop
        if A( I - 1 ) > A( I ) then
            EXCHANGE( A( I - 1 ), A( I ));
        end if;
--|     A( I ) >= A( I - 1 );
        I := I + 1;
    end loop;
```

### 2.4.4 Subprogram Annotations and Result Annotations

*Subprogram annotations* are used to describe the behavior of subprograms. They are bound to the Ada subprogram specification by the keyword **where**. They are useful in describing the input-output specifications of the subprogram. *Result annotations* are constraints on the return values of functions. A result annotation

must occur immediately within a function, but its location is otherwise not restricted. It can be in the place of an object annotation, a statement annotation, or a subprogram annotation. Result annotations are distinguished by the keyword **return** and they constrain all *return statements* within their scope. A few examples of subprogram annotations and result annotations are shown below:

```
    procedure EXCHANGE ( X, Y : in out INTEGER );
--|     where
--|         out( X = in Y ),
--|         out( Y = in X );
        -- On output the value of X is the input value of Y and vice-versa.

    procedure PUSH ( E : in ELEMENT);
--|     where
--|         out( LENGTH = in LENGTH + 1 );
        -- This procedure is from the STACK package shown earlier. In addition to illustrating subprogram
        -- annotations, this also shows how virtual functions can be used for annotation purposes. The
        -- above annotation says that the execution of PUSH causes the the value of LENGTH to increase
        -- by 1.

    function SQRT(X:FLOAT) return FLOAT;
--|     where
--|         return Y : FLOAT => Y * Y = X;
        -- The value Y returned by SQRT is such that its square is equal to the input parameter X.
```

### 2.4.5 Axiomatic Annotations

*Axiomatic annotations* (or package axioms) are constraints on package operations. They must occur in the package visible part. They are characterized by the keyword **axiom** followed by a sequence of BOOLEAN expressions which are usually quantified with respect to types defined in the package. Axiomatic annotations are *promises* which can be assumed wherever the package is visible; and they are also constraints on the implementation of the package. Algebraic specifications of abstract data types can be written as axiomatic annotations. The STACK package shown earlier is specified axiomatically in the following example:

```
    package STACK is
--:     function LENGTH return INTEGER;
        procedure PUSH ( E : in ELEMENT );
--|         where
--|             out( LENGTH = in LENGTH + 1 );
        function POP return ELEMENT;
--|         where
--|             out( LENGTH = in LENGTH - 1 );
--| axiom for all S : STACK; E : ELEMENT =>
--|     S[PUSH(E); POP] = S,
--|     S[PUSH(E)].POP = E;
    -- The above two constraints have to be satisfied by all implementations of this package, and therefore
    -- can be assumed wherever the package is visible.
    end STACK;
```

### 2.4.6 Context Annotations

A *context annotation* constrains the use of global variables within a program unit. Context annotations take the form of the keyword **limited** followed by a list of zero or more variables. It constrains the occurrences of variables declared outside of its scope (the program unit)—only those outside variables that are listed in the annotation may occur (be used) within its scope. A context annotation on the stack package mentioned earlier is shown below:

```
--| limited to INTEGER, ELEMENT;
    package STACK is
       ...
    end STACK;
```

### 2.4.7 Exception Annotations

*Exception annotations* (or propagation annotations) specify the exceptional behavior of program units. There are two different kinds of exception annotations—*strong propagation annotations*, which specify the conditions under which certain exceptions should be propagated (outside the scope of the annotation); and *weak propagation annotations*, which specify what happens when an exception is actually propagated.

#### Strong Propagation Annotations

Strong propagation annotations specify the conditions under which exceptions should be propagated. The conditions are with respect to the initial state of the scope of the annotation. If the conditions are satisfied, the scope of the annotation must be exited by propagating the specified exception. Again, examples follow with respect to the stack example:

```
    package STACK is
--:    function LENGTH return INTEGER;
       UNDERFLOW : exception;
       ...
       function POP return ELEMENT;
--|       where LENGTH = 0 => raise UNDERFLOW;
          -- If LENGTH is 0 when POP starts execution, then it must terminate by propagating the
          -- exception UNDERFLOW.
       ...
    end STACK;
```

#### Weak Propagation Annotations

Weak propagation annotations specify what happens when an exception is propagated. It specifies conditions that must be satisfied if the scope of the annotation is exited by propagating one of the specified exceptions. An example is shown below:

```
    procedure EXCHANGE(X, Y: in out INTEGER);
--|    where raise CONSTRAINT_ERROR => X = in X and Y = in Y;
       -- If the exception CONSTRAINT_ERROR is propagated out of the procedure EXCHANGE, then
       -- the parameters X and Y remain unchanged.
```

# Chapter 3

# Package Specification Methodology

## 3.1 Introduction

The major goal of this chapter is to illustrate an Anna formal specification of an Ada set package visible part. The example is designed to simply show a methodology of Anna specification, not the details of Anna. The methodology starts with an English description of the requirements of a set package specification, transitions through the Ada visible package specification, and concludes with an Anna formal specification. This is a methodology for completely specifying the behavior of Ada packages. In many cases, it is not necessary or desirable to completely specify a package; instead proving important properties of small pieces of a system can be critical. However, we will explore this methodology by delving into some details to illustrate the techniques and pitfalls involved in the process of Anna specification and development.

A secondary goal of this chapter is to teach the reader how to use the associated set of Anna tools developed at Stanford University over the past six years. This chapter aims to give readers a new way of looking at the Anna specification design and development process, synthesizing around fifteen years of research in the process.

The next section gives the natural language requirements and a normal Ada package specification. The third section presents a methodology for theoretically annotating this package along with the results of applying the Anna Specification Analyzer to the specification. The fourth section transitions this specification from a theoretical one to a practical one; a specification which can be used to check an implementation. This chapter concludes with a discussion of methods to keep as much of the specification as possible within the current Anna Transformer's subset.

## 3.2 The Abstract Data Type, Set

A *set* is a collection of elements drawn from a class of objects called the *base type*. This collection may not contain any duplicate elements. Thus, a set can be represented as an unordered collection of unique elements. A variable of type Set may represent either none, any, some, or all of the values in its base type. The elements of a set can not be selected individually. Instead, the membership operation *Is_In* is used to test for an element's presence or absence. If a set does not have any elements, it is said to be *empty*, and is denoted by the *empty_set*. The number of elements in a set is called its *cardinality*.

An element can be inserted into a set, thereby making that element a member of the resulting set. The set can be modified again by removing that element from the set. These operations will be called *Add* and *Remove* respectively. Two sets are *equal* if and only if they have the same elements as members.

The set $L$ is a *subset* of a set $R$ if every member of $L$ is a member of $R$; this will be written $L <= R$. Additionally, if $L$ is not equal to $R$, then $L$ is a *proper subset* of $R$, and is denoted $L < R$. Subset and proper subset have complementary relational operations of *superset* and *proper superset*, written $L >= R$

and $L > R$. The *union*, $L + R$, is the set whose members belong either to $L$ or to $R$ (or both), and their *intersection*, $L * R$, is the set whose members belong to both $L$ and $R$. The *set difference*, $L - R$, is the set of members of $L$ which are not in $R$. Finally, the *symmetric difference* of $L$ and $R$, denoted $L/R$, is defined by the previous operations as $(L - R) + (R - L)$.

Typical implementation strategies include linked lists and bit vectors. The two strategies have different properties, some of which limit the usefulness of the approach but benefit the implementor. Set packages implemented using linked lists have no specific maximum cardinality, however there is some uncertainty as to when the system can allocate new links during phases of its computation. This uncertainty may permit unpredictable behavior in concurrent applications. Anna can annotate these uncertainties, making them explicit in the specification.

Sets packages implemented using bit vectors have a constant maximum cardinality, which typically equals the machine word size. This restricts the usefulness of this approach, since the sets must be small. However, it is very useful and predictable, since it does not use dynamic heap allocation.

## 3.3 Ada Set Package Specification

This section presents an Ada package specification for the set abstract data type. The package is generic and forms a template from which many analogous packages can be produced without duplication of effort. It is a simple Ada package specification, written in a popular coding style, i.e. with very few informal comments. Ada package specifications were designed to separate the *syntactic* and *type semantic* definition of the interface from the actual implementation of the package body, and there is no way to define the behavioral features of the interface except with informal comments in a natural language.

```
generic
   type Base is private;
package SET_Ada is
   type Set is limited private;
   function Is_In ( E : in Base; S : in Set ) return Boolean;
      -- Membership: Returns true if the element is a member of the set.
   function Cardinality ( S : in Set ) return Counter;
      -- Returns the current number of elements in the set.
   function "=" ( L, R : in Set ) return Boolean;
      -- Equality: Returns true if the two given sets have the same state.
   procedure Copy ( S : in Set; Tx : in out Set );
      -- Copies the members of the S set over to Tx
   function Is_Empty ( S : in Set ) return Boolean;
      -- Return true if the set contains no elements.
   function Empty_Set return Set;
      -- Returns a set which contains no elements as members.
   procedure Clear ( S : in out Set );
      -- Remove all the elements (if any) from the set
   function "<=" ( L, R : in Set ) return Boolean;
      -- Subset: Returns true if the first set is a subset of the second set.
   function "<" ( L, R : in Set ) return Boolean;
      -- Proper Subset: Returns true if the first set is a proper subset of the second set.
   function "+" ( S : in Set; E : in Base ) return Set;
      -- Add: Insert an element as a member of the set.
   procedure Add ( E : in Base; S : in out Set );
      -- Insert an element as a member of the set.
   procedure Union ( L, R : in Set; Tx : in out Set );
      -- Returns a set containing the elements that are members of the first set or the second set.
```

```
    function "-" ( S : in Set; E : in Base ) return Set;
        -- Remove: Returns a set without the element as a member of the set.
    procedure Remove ( E : in Base; S : in out Set );
        -- Returns a set without the element as a member of the set.
    procedure Difference ( L, R : in Set; Tx : in out Set );
        -- Returns a set containing the elements that are members of the first set and not members of
        -- the second set.
    procedure Symmetric_Difference ( L, R : in Set; Tx : in out Set );
        -- Given two sets, form a set containing the elements that are members of the first set and are
        -- not members of the second set and the elements that are members of the second set and are
        -- not members of the first set.
    procedure Intersection ( L, R : in Set; Tx : in out Set );
        -- Given two sets, form a set containing the elements that are members of the first set and are
        -- members of the second set.
private
    -- to be defined later
end SET_Ada;
```

Even with the informal comments, it is apparent that the Ada package specification does not fully capture the meaning of the subprograms. The specification does capture the information hiding and data encapsulation semantics of this abstraction. It hides the structure of the types Set and Base by making them private types, and it encapsulates all of the subprograms associated with these two types in one place. However, it does not specify many properties which really define the meaning of the subprograms. Questions about the implementation of the package body include: Are there any exceptions propagated from the body? Is there a limit to the number of elements in a set? And if there is, what happen when this limit is exceeded? Will the procedure Remove blow up if the element is not in the set? We will use Anna to answer these questions as well as to constrain the behavior of the package body to match our assumptions; for example, that the function Empty_Set must return a set which has no elements as members.

Notice that we have presented the operations Add and Remove in two ways, each as a function and as a procedure. In the case of functions, the returned set is is the product of the given parameters, which are left unchanged. However for the procedures, the last parameter is changed and is denoted **in out** Set. This would reduce the copying of elements from one set to another which may occur in function based operations. We use this difference between procedure based operations and function based operations to show a slightly different style of Anna subprogram specification.

## 3.4 Theory Package Methodology

This section presents a methodology of writing Anna specifications for Ada package specifications. The exact details of the methodology will depend on what the package will implement as well as the experience of the individual doing the specification. Specifications may consist of:

- a collection of subprograms.
- an abstract data type, with values stored in the private or limited types.
- an abstract data object, with values stored in the package state.
- a generic abstract data type.
- a generic abstract data object.
- a combination of the above.

In particular, we will continue with our specification of a set package, a generic abstract data type. We consider the specification of the other package types to be a simple extension to the method we will present.

While the specification methodology is presented in basically a straight line fashion, there are implied feedback loops. Writing a subprogram annotation or axiom may require jumping back to define a new virtual function, a new exception, or a new enumeration literal before continuing.

### 3.4.1 Write the Ada Package

We have just performed the first step in our development methodology: write the Ada package visible part. This part remains basically unchanged throughout; we just make extensions and reorderings to it. The ordering of the declarations in the package visible part is normally loose — the only Ada restriction is that a type or subtype must be declared (perhaps incompletely) before being referenced in another (sub)type or subprogram declaration. Exception declarations can be placed anywhere since they are not referenced in any other Ada declaration.

Adding Anna specifications will place additional restrictions on the ordering of the Ada declarations. The annotations must occur after the definitions of all of the entities that are used in the annotations. Notably, this now restricts the placement of exception declarations, since they must now appear before any reference in propagation annotations.

The suggested order of the declarations in the package visible part is:

- (sub)types, constants, and objects
- exceptions
- virtual basic concept functions
- visible basic concept functions
- annotated virtual functions
- annotated visible subprograms
- package initial axioms
- package axioms
- package private part

### 3.4.2 Basic Concept Functions

We will now develop a set of functions of a set package that capture the intrinsic features of the abstract data type, called basic concepts. These basic concepts will be used to specify the other subprograms. For example, the basic concepts of a set package could be membership, equality, cardinality, and overflowed. The basic concepts will be written as virtual or visible functions. The virtual functions are only visible in annotations in the package visible part (and for future annotations in the package body). The virtual functions can be compared to peepholes into the hidden data structures in the package body.

Virtual functions make internal package conditions available for use in the subprogram annotations. This is very important when the original requirements define a hidden internal condition that can change the behavior of a subprogram. In the set package requirements, the condition that the system may overflow upon an Add if more heap memory allocation is not possible. The system running out of memory is an internal condition that may affect a change in the visible subprograms, and is therefore a basic concept. A virtual function, Overflowed, is required to make the internal condition visible.

The basic concept functions are intrinsic to the abstract data type and can not be directly annotated by out-state or result annotations. They do not change the package state of the package; they only report on

internal package data or conditions. A basic concept function can be annotated with object annotations (for input or pre-conditions), or strong and weak propagation annotations (for exceptions). This is because a function which has a result annotation is specified using other concepts. Such functions can not be understood fully unless the concepts on which they are specified are fully understood.

A recursive result annotation is specified in terms of another subprogram, which is in turn specified in terms of the original subprogram. Recursive result annotations are not allowed, as they present a major technical problem for the Anna toolset. The Anna transformer introduces checking calls, which upon execution will form a cycle and recurse infinitely. Therefore, subprograms should be stratified to eliminate the possibility of infinite recursion.

In general, the set of basic concepts should be as small as possible without complicating the package specification. We define one set of concepts to be smaller than another if and only if it is a subset of the other. Furthermore, a given specification may not have a unique minimal set of basic concepts. We are simply trying to keep the number of basic concepts small. This is because we are assuming that that we understand the basic concept and thus can use it to understand other subprograms.

In the case of the set package, we may consider Is_In and Equality to be basic concepts. If we leave both unspecified, they will both be basic. However, we define equality using Is_In, and then only Is_In will be basic. Thus, the general rule of thumb would imply that the only basic concept needed is Is_In. Since the subprogram Is_In is taken as a basic concept, we define the equality relation on sets based on our concept of membership.

```
-- Basic Concept Functions
--:    function Overflowed return Boolean;
      function Is_In ( E : in Base; S : in Set ) return Boolean;
      function Cardinality ( S : in Set ) return Natural;

-- Nonbasic Concept Function
      function "=" ( L, R : in Set ) return Boolean;
--|     where
--|       return for all E : Base => Is_In(E, L) <-> Is_In(E, R);
          -- The annotation states that the equality relation returns true if and only if every element
          -- which is a member of L is also a member of R and every element which is not a member of
          -- L is not a member of R.
          -- By Anna semantic rules, this annotation holds trivially for undefined values of Base.
```

### 3.4.3 Subprogram Annotation

For each visible subprogram in the package, the following questions should be answered by the requirements and converted into Anna subprogram annotations. If the answer is not in the requirements, this may indicate a missing or incorrect requirement. Each subprogram annotation must specify all of the changes that the subprogram will make to the package state (if possible).

**Package State Conditions** What constraints on the package state must be true when the subprogram is called? Use: *Object Annotations*

**In Parameters** What are the constraints on the **in** and **in out** parameters (other than normal Ada type constraints)? Use: *Object Annotations*

**Exceptions** What package defined exceptions are propagated by this subprogram? What predefined Ada exceptions are explicitly propagated by this subprogram? How has the resulting package state changed after each exception is propagated? Use: *Strong Propagation Annotations* and *Weak Propagation Annotations* (reserved word **raise**)

**Function Result** What is the result of the function? Use: *Result Annotation* (reserved word **return**)

**Out Parameters** What are the returned values of **out** and **in out** parameters? Use: *Out Annotations* (reserved word **out**)

**Package State Changes** What are the changes to the package state when the subprogram returns normally? Use: *Out Annotations* and / or *Result Annotations* (reserved words **out** and **return**)

### Is_Empty, Empty_Set and Clear

Let's examine empty sets in more detail, with respect to the annotation topics that a good subprogram specification should address. The annotations on empty sets, below, constrain a set to be empty when the set has no elements as members. This informal explanation specifies most of the semantics of empty sets, excluding the semantics of exceptions raised. We will delay this discussion of exceptional behavior until later in this chapter.

The function Is_Empty is easily formally specified by checking every possible base object and make sure it is not a member of the given set. The function result is simply a universal quantification over the domain of elements in the base, derived directly from the informal description. There are no package state changes.

We continue similarly with the specification of Empty_Set and Clear. The other subprograms can reuse the specification for Is_Empty, as shown below. An interesting point is the choice of making Empty_Set a function or a constant. Both declarations can be used, and they have the same external semantics, because the value returned by the function will be a constant. However, as a function it gives us an opportunity to use both a subprogram and axiomatic annotations.

Also, we would like to specify the default initial value of a set. The initial package axioms define the package state immediately after the package body has been elaborated. The initial axioms should define the initial result values for all of the basic concept functions, both virtual and visible. The axioms should also define the initial values of all private and limited private type objects. Used here, the Anna attribute 'Initial is an attribute of any private type which denotes the default initial value of objects of the type. The axiom defines every set to be initially equal to the Empty_Set, where this equality is not defined yet!

```
        function Is_Empty ( S : in Set ) return Boolean;
--|       where
--|         return for all E : Base => not Is_In(E, S);
            -- Returns true if there are no elements in the set.
        function Empty_Set return Set;
--|       where
--|         return S : Set => Is_Empty( S );
            -- Returns a set which must be empty.
        procedure Clear ( S : in out Set );
--|       where
--|         out( Is_Empty( S ));
            -- After completion the returned set must be empty.
--| axiom
--|   Is_Empty( Set'Initial ),
--|   for all E : Base => not Is_In(E, Empty_Set);
      -- Appropriate whether Empty_Set is a function or a constant.
```

### Subset and Proper Subset

The abstraction defines $L$ to be a subset of $R$ if every element which is member of $L$ is also a member of $R$. This is easily expressed in Anna as an implication expression. The subset annotation states that $L <= R$ if and only if every element which is a member of $L$ is also a member of $R$.

The abstraction defines the $L$ to be a proper subset of $R$ if $L$ is a subset of $R$ and additionally $L$ is not equal to $R$. This formal definition uses two previously defined concepts in the abstraction to define another. In Anna, we do an analogous formal definition. Since we have already defined in Anna the specification of equality and subset, we can use them to specify proper subset. This is expressed as a conjunction of $L$ being the subset of $R$ and $L$ being not equal to $R$.

```
        function "<=" ( L, R : in Set ) return Boolean;
--|       where
--|         return for all E : Base => Is_In(E, L) -> Is_In(E, R);
            -- Return True if the first set is a subset of the second set.
        function "<" ( L, R : in Set ) return Boolean;
--|       where
--|         return (L <= R) and L /= R;
            -- Return True if the first set is a proper subset of the second set.
```

### Add and Union

The operations Add and Union are very similar; in fact one can think of Add as an overloading of Union, where the element being added is a singleton set. Union takes two sets and forms a set containing the elements that are members of the first set or the second set. The annotation for Union states that every element which is a member of the returned set, $Tx$, is either a member of the given set $L$ or a member of the given set $R$. For the Add operation the set being returned, the out–value of S, has every member of the in–value of S plus the new element, $E$. This annotation implies that if the new element $E$ is already a member of set $S$, then $S$ does not change.

```
        function "+" ( S : in Set; E : in Base ) return Set;
--|       where
--|         return R : Set =>
--|           for all A : Base =>
--|             ( Is_In(A, S) or A = E ) <-> Is_In(A, R);
            -- Insert an element as a member of the set.
        procedure Add ( E : in Base; S : in out Set );
--|       where
--|         out( for all A : Base =>
--|                 ( Is_In(A, in S) or A = E ) <-> Is_In(A, S));
            -- Insert an element as a member of the set.
        procedure Union ( L, R : in Set; Tx : in out Set );
--|       where
--|         out( for all E : Base =>
--|                 ( Is_In(E, L) or Is_In(E, R)) <-> Is_In(E, Tx));
         -- Returns a set containing the elements that are members of the first set or the second set.
```

### Remove and Set Difference

The operation Remove modifies the set making the element no longer in the set. The operation Set Difference forms a set containing the elements that are members of the first set and not members of the second set.

These annotations are about the same length as Add and Union. We want to reuse our previous annotations to derive new annotations, which are shorter than they would be otherwise. Here we introduce the style of arranging the subprograms in strata. Formally, the subprograms are stratified if whenever there is a subprogram specification for $p$ that uses another subprogram $q$, there is no path in the dependency graph

from $q$ to $p$. Intuitively, a subprogram's specification can only use a subprogram that lies in a lower stratum. Basic concepts are always placed in the lowest stratum.

We will exemplify this process by doubly specifying the operations Remove and Set Difference. The first specification will be similar to the previous specifications using the basic concepts. The second specification will reuse any previous specifications of subprograms, creating a stratified set of subprograms.

```
        function "-" ( S : in Set; E : in Base ) return Set;
--|       where
--|         return R : Set =>
--|           ( for all A : Base =>
--|               ( Is_In(A, S) and not A = E ) <-> Is_In(A, R))
--|           and ( S = R + E );
            -- Remove: Returns a set without the element as a member of the set.
        procedure Remove ( E : in Base; S : in out Set );
--|       where
--|         out( for all A : Base =>
--|                 ( Is_In(A, in S) and not A = E ) <-> Is_In(A, S)),
--|         out( S = "-"( in S, E ));
            -- Returns a set without the element as a member of the set.
        procedure Difference ( L, R : in Set; Tx : in out Set );
--|       where
--|         out( for all E : Base =>
--|                 ( Is_In(E, L) or Is_In(E, R)) <-> Is_In(E, Tx)),
--|         out( Tx = L + R );
            -- Returns a set containing the elements that are members of the first set and not members of
            -- the second set.
```

The strata so far are:
Stratum 1: The basic concept functions: Is_In, Overflow, and Cardinality.
Stratum 2: Equality, Is_Empty, Subset, Add, and Union
Stratum 3: Empty_Set, Clear, Proper Subset, Remove and Set Difference

### Intersection and Symmetric Difference

Intersection returns a set containing the elements that are members of the first set and are members of the second set. The first annotation states that an element E is a member of the $L * R$, if and only if E is a member of $L$ and a member of $R$. The second part of this annotation describes the resulting set as equivalent to a combination of previously defined functions. Symmetric difference is specified similarly.

```
        procedure Intersection ( L, R : in Set; Tx : in out Set );
--|       where
--|         out( for all E : Base =>
--|                 ( Is_In(E, L) and Is_In(E, R)) <-> Is_In(E, Tx)),
--|         out( Tx = (L + R) - (S - T) - (T - S));
            -- Given two sets, form a set containing the elements that are members of the first set and are
            -- members of the second set.
```

```
        procedure Symmetric_Difference ( L, R : in       Set;
                                         Tx   : in out   Set);
--|       where
--|         out( for all E : Base =>
--|                    ( Is_In(E, L) xor Is_In(E, R)) <-> Is_In(E, Tx)),
--|         out( Tx = (L - R) + (R - L));
          -- Given two sets, form a set containing the elements that are members of the first set and are
          -- not members of the second set and the elements that are members of the second set and are
          -- not members of the first set.
```

### Cardinality

The cardinality of a set is the number of elements which are currently members of the set. The specification for cardinality is much different from previous specifications. This specification must in some sense "count" the number of elements in a set. This specification begins with a base case, for the Empty_Set, and then specifies the inductive step. The base case is very simple, since we just need to state that the cardinality of the Empty_Set is zero.

The inductive step assumes the cardinality for a set, $T$, is known. Then we notice that set $S$ is equal to adding element $E$, which is not a member of $T$, to set $T$. The cardinality for $S$ is simply one more than the cardinality of set $T$. Now, we can specify the inductive step two ways. We can use only the basic concept functions, or we can use higher stratum subprograms. Also, we can specify Cardinality with either subprogram annotations or with package axioms. There is of course no problem with "over-specifying" with all of these specifications, as presented below.

```
        function Cardinality ( S : in Set ) return Natural;
--|       where
--|         return N : Natural =>
--|           if S = Empty_Set then
                  -- Base Case
--|               N = 0
--|           else
                  -- Inductive Step
--|               ( exist T : Set; E : Base =>
--|                   ( Is_In(E, S) and not Is_In(E, T) and S = T + E ) and
--|                   ( for all L : Base =>
                        E /= L -> ( Is_In(L, S) <-> Is_In(L, T))) and then
--|                   N = Cardinality(T) + 1 )
--|           end if;
          -- Returns the current number of elements in the set.
```

```
--| axiom for all S, T : Set =>
          -- Base Case:
--|             Cardinality( Empty_Set ) = 0,
          -- Inductive Step, using only Is_In:
--|             ( exist E : Base =>
--|               Is_In(E, S) and not Is_In(E, T) and
--|               ( for all L : Base => E /= L ->
--|                   ( Is_In(L, S) <-> Is_In(L, T))))
--|             -> Cardinality( S ) = Cardinality(T) + 1,
          -- Inductive Step, using Is_In and Add:
--|             ( exist E : Base =>
--|               not Is_In(E, T) and S = T + E )
--|             -> Cardinality(S) = Cardinality(T) + 1;
```

### 3.4.4 Iterators

Iterators permit a sequence of statements to be executed zero or more times on all parts of an object. An iterator must permit non-destructive "visit" to each part of the object. Basically, an iterator is manifest as an operation (or set of operations) that is exported from a component specification. We can view an iterator as another object that traverses the structure of the encapsulated object.

For simple unencapsulated types such as arrays, a *for loop* that indexes each component of the array is an iterator. However, for the set abstraction, we do not have an index on which to base a loop. Typically, the using program will need to loop, performing a sequence of statements or operations, on each element of a set.

The set package therefore provides enumeration functions for sets. Each time the enumeration function is called, it returns the "next" component. Obviously, the iterator must remember which elements have already been returned. It can use the state of the sets package or define an iterator type to do this.

Booch[1] describes two iterators that reflect two different approaches to iteration. This difference is how much of the abstraction we want to expose outside the encapsulation. In the active approach, we expose the iterator as a collection of primitive operations, but in the passive approach, we export only a single operation.

#### Active Iterator

An active iterator is an abstract data type with the following four operations: 1) an initializer which associates the iterator with a set; 2) a value function which returns the current element in the set denoted by the iterator; 3) a increment procedure which advances the iterator to the next element in the set; 4) a finished function that returns true if the iterator has visited all of the members of the set.

We introduce the virtual function Set_Of to return the set associated with the iterator. In this way we can map from an iterator to a set, to check that it is popping off the correct value.

```
      type Iterator is limited private;
--:   function Set_Of ( It : in Iterator ) return Set;
      procedure Initialize ( It : in out Iterator; S : in Set );
--|     where
--|       out( Set_Of( It ) = S );
      function Value_Of( It : in Iterator ) return Base;
--|     where
--|       return E : Base => Is_In(E, Set_Of( It ));
      procedure Get_Next( It : in out Iterator );
--|     where not Is_Empty( Set_Of( It )),
--|       out( Set_Of( It ) = Set_Of( in It ) - Value_Of( in It ));
```

```
        function Is_Done ( It : in Iterator ) return Boolean;
--|       where
--|         return Is_Empty( Set_Of( It ));
--| axiom Is_Done( Iterator'Initial );
```

Iteration works as follows. Suppose we want to enumerate the elements of some set A_Set. First we initialize the iteration to A_Set. Then we call Get_Next repeatedly, getting a new member of A_Set from Value_Of. We loop until Is_Done becauses true. Is_Done becomes true only after all the elements of A_Set have been visited. For example:

```
I : Iterator; A_Set : Set;

procedure Visit ( E : in Base );
...
Initialize( I, A_Set );
while not Is_Done( I ) loop
  Visit( Value_Of( I ));
  Get_Next( I );
end loop;
```

**Passive Iterator**

This is an interesting subprogram to attempt to specify. Since the in and out states are equivalent with respect to our observable state, nothing can be specified. Therefore, passive iterators are not within the concepts specifiable using this methodology.

```
generic
   with procedure Process(E : in Base; Continue : out Boolean);
procedure Iterate ( S : in Set );
```

### 3.4.5 Package Specification Analysis

Now that we have an Anna specification that includes formal subprogram and axiomatic specifications, we would like to perform a critical analysis to determine the correctness and the completeness of our specifications. That is, we would like to make sure that it will constrain an implementation in the way we intended.

The Anna Package Specification Analyzer of the Anna-I Toolset[10], is a tool for analyzing and debugging the visible part of Anna package specifications. It uses logical deduction embedded in a user interface that allows symbolic execution of subprograms based on their specifications, and analysis of how these executions affect the state of the package and its environment.

While in general the Specification Analyzer understands axiomatic specifications, some forms of axioms are difficult to use in resolution theorem provers. Some transformations of annotations to equivalent alternative forms may be required to test an Anna package satisfactorily.

For instance, recall the axioms defined for the Cardinality subprogram, using Is_In and "+":

```
--| axiom for all S, T : Set =>
--|             Cardinality( Empty_Set ) = 0,
--|             ( exist E : Base =>
--|               not Is_In(E, T) and S = T + E )
--|             -> Cardinality(S) = Cardinality(T) + 1;
```

Given a specific set S, which has been built through various calls to subprograms in the set package, it is difficult to deduce automatically from the above axiom what the cardinality of S is; in general, a prover would have to iterate over all possible sets, for values of the quantified variable T, and all possible Base values, for values of E.

However, and equivalent axiom can be written, based on the fact that every set has been built through calls to the set constructors:

```
--| axiom for all S : Set; E : Base =>
--|             Cardinality( Empty_Set ) = 0,
--|             Cardinality( S + E ) =
--|               if Is_In(E, S) then Cardinality(S)
--|               else Cardinality(S) + 1 end if,
--|             Cardinality( S - E ) =
--|               if Is_In(E, S) then Cardinality(S) - 1
--|               else Cardinality(S) end if;
```

Every set is equivalent to an expression consisting of the constructors that built it, such as:

```
((Empty_Set + E1) + E2) - E1
```

The Specification Analyzer is able to determine what constructor expression a particular set is equivalent to, and determine its cardinality by "deconstructing" the set based on the above axioms.

## 3.5 From Theory to Practice

Looking back over our specifications, we realize that we have really created a theory about sets. But if we really wanted to use this as an actual Ada package, a number of things are missing.

One thing missing is exception handling. The annotations presented so far do not constrain the subprograms during exceptional situations. To specify these cases, we must examine each subprogram in turn. In doing so, we notice that some subprograms should never raise exceptions, while others should raise a exception given certain inputs, and still others may raise several exceptions, each one for a different reason.

### 3.5.1 Never Raise an Exception

Some subprograms are guaranteed never to raise an exception. For example, all of the basic concepts should not raise any exceptions. The Anna construct for specifying this is very simple. The subprograms annotations for cardinality, equality, subset and proper subset are extended just like Is_In is extended below.

```
      function Is_In ( E : in Base; S : in Set ) return Boolean;
--|     where raise others => False;
          -- No exceptions are raised by this function.
```

### 3.5.2 Exceptions: Is_In_Set and Is_Not_In_Set

Another typical exceptional situation is when we try to add an element into a set, but the element is already a member of the set. This situation is in conflict with the style of Union which does nothing when there are duplicate elements in its input sets. However, we reject Union's approach, choosing instead to apply the more primitive semantics. In this way, we differentiate Add from the semantics of Union. Thus, when we try to add an element which is already a member of the set, the exception Is_In_Set shall be raised. Also, only this exception can be raised by the subprogram. Therefore, if any exception is raised by Add, then

the element must be a member of the given Set, thus implying that the exception Is_In_Set must have been raised.

```
        Is_In_Set : exception

        function "+" ( S : in Set; E : in Base ) return Set;
--|       where Is_In(E, S)  => raise Is_In_Set,
--|             raise others => Is_In(E, S);
            -- Only the exception Is_In_Set can be raised, and it is raised when the element to be added
            -- is already in the given set.
--|         return R : Set =>
--|           for all A : Base => ( Is_In(A, S) or A = E ) <-> Is_In(A, R);
        procedure Add ( E : in Base; S : in out Set );
--|       where Is_In(E, S)  => raise Is_In_Set,
--|             raise others => Is_In(E, S);
--|         out( for all A : Base => ( Is_In(A, in S) or A = E ) <-> Is_In(A, S));
        procedure Union ( L, R : in Set; Tx : in out Set );
--|       where raise others => False;
--|         out( for all E : Base => ( Is_In(E, L) or Is_In(E, R)) <-> Is_In(E, Tx));
```

A similar argument holds for Remove and Set Difference. If the element to be removed is not a member of the given set, then raise the exception Is_Not_In_Set. Additionally, if an exception is raised, then the element is not a member of the given set, thereby implying that the exception Is_Not_In_Set was raised.

### 3.5.3 Exceptions: Overflow and Set_Full

Any operation which returns a set may raise one of the two exceptions, Overflow or Set_Full. The Overflow exception is raised whenever the system has exhausted its memory and can no longer allocate more memory for the set. To properly specify this exception, we need to introduce a new basic concept to tell us when the memory has run out. We introduce an Anna virtual function Overflowed to return true if and only if the memory of the system has run out. This exception can occur in every subprogram which extends a set.

Notice that Overflow is a property of the state of the machine. We may have a situation where a particular variable of type Set can not contain any more elements. Here we assume that some implementations of this abstraction will constrain the maximum size of a variable of type Set. This constraint limits the number of elements that can be members of a set at one time. We introduce the virtual function Is_Full which returns true if and only if the given set can not contain any more new members. This can only happen in Add and Union.

For Add it is fairly simple to understand exactly when the exception Set_Full is raised. We raise the exception Set_Full when the given set is full and the element to be added to it is not already a member. However, Union is slightly more difficult to specify exactly. Since we don't know how big the created set will be until we evaluate it, we can't exactly tell if it will be too big. Here we will introduce a further assumption that the limit on the maximum cardinality of a set is a constant for all sets.

```
        Overflow, Set_Full : exception

--:     function Overflowed return Boolean;
--|       where raise others => False,
            -- Returns true if the system has run out of memory.
--:     function Maximum_Cardinality return Natural;
--|       where raise others => False,
            -- Returns the maximum cardinality of any set.
            -- This assumes the maximum cardinality is a finite and constant for all sets.
```

```
--:   function Is_Full ( S : in Set ) return Boolean;
--|     where raise others => False,
--|       return Cardinality( S ) = Maximum_Cardinality;
          -- Return true if the set can not contain any more elements.
      function "+" ( S : in Set; E : in Base ) return Set;
--|     where Is_In(E, S) => raise Is_In_Set,
--|           Is_Full(S) and not Is_In(E, S) => raise Set_Full,
--|           Overflowed and not (Is_Full(S) or Is_In(E, S)) => raise Overflow,
--|           raise others => Is_Full(S) or Is_In(E, S) or Overflowed,
--|       return R : Set =>
--|         for all A : Base =>
--|             ( Is_In(A, S) or A = E ) <-> Is_In(A, R);
      procedure Union ( L, R : in Set; Tx : in out Set );
--|     where Cardinality(L) + Cardinality(R)
--|             - Cardinality(L*R) > Maximum_Cardinality => raise Set_Full,
--|           not ( Cardinality(L) + Cardinality(R)
--|             - Cardinality(L*R) > Maximum_Cardinality ) and
--|           Overflowed  => raise Overflow,
--|       out( for all E : Base =>
--|               ( Is_In(E, L) or Is_In(E, R)) <-> Is_In(E, Tx));
```

Suppose we want to define an alternative semantics for a different kind of set, one in which the individual sets have no size limit, but you still might run out of heap space. We can use axioms to "tune" a particular implementation's memory management.

```
--| axiom for all S : Set => not Is_Full( S );
--| axiom exist State : Set'Type => State.Overflowed;
```

## 3.6 Conversion to Transformer Subset

We have now finished our Anna "theoretical" specification of the set package. We call it theoretical because we have used constructs which are not currently supported by the Anna Transformer in the Anna-I toolset. If we wish to use the current toolset to provide a self-checking implementation of the package, we must modify these constructs.

### 3.6.1 Quantifiers

The main problem is the use of quantifiers. We have used universal and existential quantifiers to traverse the domain of the elements making assertions about the membership of each element in a set. It is important to note that the only reason we have the quantifiers is to test some property based upon membership in one or more sets.

We notice that the quantifier can be replaced by an iterator, if the iterator iterates through all elements in one or more sets, testing for the property. An abstract loop on an iterator for a set really does four things: 1) it can select an arbitrary element, 2) it can remove an element from the set, and 3) it can tell when we have visited all of the elements of the set. We notice that the last case is just when the resulting set is empty. Thus, we would like to introduce three new basic functions that can fulfill these requirements. We re-declare Is_Empty, this time without an annotation. Additionally, we declare two new functions, An_Element and Sub. Sub should not be confused with either Remove subprogram. Sub never raises an exception, while Remove sometimes does. Also, these new subprograms are very similar in concept to the active iterator. In some sense, this is also why they are basic concepts. Since it is very difficult to express the semantics of an

active iterator and stay within the checkable subset, we also can not express these new basic concepts more fundamentally.

```
        function Is_Empty ( S : in Set ) return Boolean;
            -- Returns true if the given set has no members.
--:     function An_Element ( S : in Set ) return Base;
--|       where not Is_Empty( S ),-- When called, S must not be empty.
--|             raise others => False;
            -- Returns an arbitrary item in the set.
--:     function Sub ( S : in Set; E : in Base ) return Set;
--|       where raise others => False;
            -- Remove an item as a member of the set, if it is one.
```

We create a new lowest stratum and place Is_Empty, An_Element, and Sub in this stratum. Now, since we have introduced three new basic concept functions, we compare them to our old basic concept function Is_In. We notice that we can specify Is_In with the specifications of the new basic concepts functions.

```
        function Is_In ( E : in Base; S : in Set ) return Boolean;
--|       where raise others => False,
--|             return ( not Is_Empty( S )) and then
--|                    (( An_Element( S ) = E ) or else
--|                     ( Is_In(E, Sub( S, An_Element( S )))));
```

We should go back and examine all of the subprograms annotations which used universal or existential quantification, and see if we can respecify them with our new basic concept functions. For example, equality and cardinality:

```
        function "=" ( L, R : in Set ) return Boolean;
--|       where raise others => False,
--|             return if Is_Empty( L ) then Is_Empty( R )
--|                    else Is_In( An_Element( R ), T ) and then
--|                         Sub(L, An_Element( L )) = Sub(R, An_Element( L )),
--|                    end if;
        function Cardinality ( S : in Set ) return Natural;
--|       where raise others => False,
--|             return if Is_Empty(L) then 0
--|                    else 1 + Cardinality( Sub(S, An_Element( S )))
--|                    end if;
```

Finally, we add labels to our formal annotations, such as <<Anno_Label>>. Named annotations are useful in debugging[3].

### 3.6.2 Transformable Set Package Specification

```
-- Description:
-- An Anna specification of a set package which is within the transformer subset
-- Author:
-- John J. Kenney
-- Source:
-- Set_Simmple_Sequential_Unbounded_Unmanaged_Iterator
-- Software Components With Ada, Grady Booch
```

```
    generic
        type Base is private;
    package Set_Package is
        type Set is limited private;

        Item_Is_In_Set,
        Item_Is_Not_In_Set,
        Overflow : exception;

-- Basic Concept Functions
-- The next four functions, Overflowed, Is_Empty, Sub and An_Element, are the basic concepts used in
-- this package specification.

--:     function Overflowed return Boolean;
--|     <<Overflowed_Spec>>
--|     where raise others => false;
            -- Returns true if the system has run out of memory.

        function Is_Empty ( S : in Set ) return Boolean;
--|     <<Is_Empty_Spec>>
--|     where raise others => false;
            -- Returns true if the given set has no members.

--:     function Sub ( S : in Set; E : in Base ) return Set;
--|     <<Remove_Fn_Spec>>
--|     where raise others => false;
            -- Remove an item as a member of the set, if it is one.

--:     function An_Element ( S : in Set ) return Base;
--|     <<An_Element_Spec>>
--|     where Is_Empty( S )  => raise Item_Is_Not_In_Set,
--|           raise others   => Is_Empty( S );
            -- Returns an arbitrary item in the set.

-- Selectors

        function Is_In ( E : in Base; S : in Set ) return Boolean;
--|     <<Is_In_Spec>>
--|     where raise others => false,
--|         return ( not Is_Empty( S )) and then
--|                (( An_Element( S ) = E ) or else
--|                  ( Is_In( E, Sub( S, An_Element( S )))));
            -- Returns true if the given item is a member of the set.

        function Cardinality ( S : in Set ) return Natural;
--|     <<Cardinality_Spec>>
--|     where raise others => false,
--|         return if Is_Empty( S ) then 0
--|                else 1 + Cardinality( Sub( S, An_Element( S )))
--|                end if;
            -- Return the current number of items in the set.
```

```
        function "=" ( S, T : in Set ) return Boolean;
--|      <<Equal_Symbol_Spec>>
--|      where raise others => false,
--|           return if ( Is_Empty( S )) then Is_Empty( T )
--|                  else Is_In( An_Element( S ), T ) and then
--|                  Sub( S, An_Element( S )) = Sub( T, An_Element( S ))
--|                  end if;
             -- Returns true if the two given sets have the same state.

         function "<=" ( S, T : in Set ) return Boolean;
--|      <<Subset_Spec>>
--|      where raise others => false,
--|           return Is_Empty(S) or else
--|                  ( Is_In( An_Element( S ), T) and then
--|                    Sub( S, An_Element( S )) <= T );
             -- Return true if the first set is a subset of the second.

        function "<" ( S, T : in Set )  return Boolean;
--|      <<Proper_Subset_Spec>>
--|      where raise others => false,
--|           return ( S <= T ) and then S /= T;
             -- Return true if the first set is a proper subset of the second set.

-- Constructors

         function Empty_Set return Set;
--|      <<Empty_Set_Spec>>
--|      where raise others => false,
--|           return S : Set => Is_Empty( S );
             -- Returns true if the given set has no members.

        procedure Copy ( From : in Set; Tx: in out Set );
--|      <<Copy_Spec>>
--|      where Overflowed    => raise Overflow,
--|            raise others => Overflowed,
--|           out( From = Tx );
             -- Copy the items from one set to another.

        function "-" ( S : in Set; E : in Base ) return Set;
--|      <<Remove_Fn_Spec>>
--|      where not Is_In(E, S) => raise Item_Is_Not_In_Set,
--|            raise others => not Is_In(E, S),
--|           return Sub( S, E );
             -- Remove an item as a member of the set, if it is one.

        function "+" ( S : in Set; E : in Base ) return Set;
--|      <<Add_Fn_Spec>>
--|      where Is_In(E, S) => raise Item_Is_In_Set,
--|            Overflowed and not Is_In(E, S) => raise Overflow,
--|            raise others => Is_In(E, S) or Overflowed,
--|           return R : Set => ( S = R - E ) and Is_In(E, R);
             -- Insert an item as a member of the set.
```

```
      procedure Add ( E : in Base; S : in out Set );
--|   <<Add_Spec>>
--|   where Is_In(E, S) => raise Item_Is_In_Set,
--|         Overflowed and not Is_In(E, S) => raise Overflow,
--|         raise others => Is_In(E, S) or Overflowed,
--|       out( S = "+"( in S, E ));
        -- Insert an item as a member of the set.

      procedure Remove ( E : in Base; S : in out Set );
--|   <<Remove_Spec>>
--|   where not Is_In(E, S) => raise Item_Is_Not_In_Set,
--|         raise others    => not Is_In(E, S),
--|       out( S = in S - E );
        -- Remove an item as a member of the set, if it is one. This is fully described in the axioms with
        -- respect to add.

--:   function "+" ( L, R : in Set ) return Set;
--|   <<Union_Fn_Spec>>
--|   where raise others => false,
--|       return S : Set =>
--|           if ( Is_Empty( L )) then S = R
--|           elsif ( Is_Empty( R )) then S = L
--|           elsif ( Is_In( An_Element( L ), R )) then
--|                 S = ( L - An_Element( L )) + R
--|           else S = ( L - An_Element( L )) + ( R + An_Element( L ))
--|           end if;
        -- Given two sets, form a set containing the items that are members of the first set or the second
        -- set.

      procedure Union ( L, R : in Set; Tx : in out Set );
--|   <<Union_Spec>>
--|   where Overflowed => raise Overflow,
--|       out( Tx = L + R );
        -- Given two sets, form a set containing the items that are members of the first set or the second
        -- set.

--:   function "-" ( L, R : in Set ) return Set;
--|   <<Difference_Fn_Spec>>
--|   where Overflowed => raise Overflow,
--|         raise others => Overflowed,
--|       return S : Set =>
--|           if Is_Empty( R ) then S = L
--|           elsif ( Is_In( An_Element( R ), L )) then
--|                   S = ( L - An_Element( R )) - ( R - An_Element( R ))
--|           else S = L - ( R - An_Element( R ))
--|           end if;
```

```
      procedure Difference ( L, R : in Set; Tx : in out Set );
--|   <<Difference_Spec>>
--|   where Overflowed => raise Overflow,
--|         raise others => Overflowed,
--|       out( Tx = L - R );
          -- Given two sets, form a set containing the items that are members of the first set and not members
          -- of the second set.

--:   function "*" ( L, R : in Set ) return Set;
--|   <<Intersection_Fn_Spec>>
--|   where Overflowed    => raise Overflow,
--|         raise others => Overflowed,
--|       return if ( Is_Empty( L )) then Empty_Set
--|              elsif ( Is_In( An_Element( L ), R )) then
--|                     "+" ( "*"( L - An_Element( L ), R ), An_Element( L ))
--|              else ( L - An_Element( L )) * R
--|              end if;

      procedure Intersection ( L, R : in Set; Tx : in out Set );
--|   <<Intersection_Spec>>
--|   where Overflowed    => raise Overflow,
--|         raise others => Overflowed,
--|       out( Tx = "*"( L, R ));
          -- Given two sets, form a set containing the items that are members of the first set and the second
          -- set.

      procedure Clear ( S : in out Set );
--|   <<Clear_Spec>>
--|   where raise others => false,
--|       out( Is_Empty( S ));
--|       -- Remove all the items (if any) from the set and make the set empty.

-- Iterators

   -- Passive Iterators
      generic
         with procedure Process ( E        : in      Base;
                                  Continue :    out  Boolean );
      procedure Iterate ( S : in Set );

   -- Active Iterators
      type Iterator is limited private;
      Iterator_Error : exception;

--:   function Set_Of ( it : in Iterator ) return Set;

      procedure Initialize ( it : in out Iterator; S : in Set );
--|   <<Initialize_Spec>>
--|   where
--|       out( S = Set_Of( it ));
```

```
      function Value_Of ( it : in Iterator ) return Base;
--|    <<Value_Of_Spec>>
--|    where not Is_Empty( Set_Of( it )),
--|         return E : Base => Is_In( E, Set_Of( it ));

      procedure Get_Next ( it : in out Iterator );
--|    <<Get_Next_Spec>>
--|    where not Is_Empty( Set_Of( it )),
--|         out( Set_Of( it ) = Set_Of( it ) - Value_Of( it ));

      function  Is_Done ( it : in Iterator ) return Boolean;
--|    <<Is_Done_Spec>>
--|    where
--|         return Is_Empty( Set_Of( it ));

--| axiom for all E : Base => not Is_In( E, Empty_Set );
--| axiom Set'Initial = Empty_Set;
--| axiom for all S, T : Set =>
--|             Cardinality( Empty_Set ) = 0,
--|             (exist E : Base =>
--|                 Is_In(E, S) and
--|                 not Is_In(E, T) and
--|                 (for all L : Base =>
--|                     E /= L -> ( Is_In(L, S) <-> Is_In(L, T))))
--|             -> Cardinality( S ) = Cardinality( T ) + 1;

   private
      type Set_Rec;
      type Set_Ptr is access Set_Rec;
      type Set is new Set_Ptr;
      type Iterator_Rec;
      type Iterator is access Iterator_Rec;
   end Set_Package;
```

# Chapter 4

# Abstract Data Types

This chapter contains a number of abstract data types (ADTs), specified as Ada packages and annotated with Anna. Each abstract data type is implemented as a package that exports a private type, and a set of operations on that type. Typically, the package implements a common software data structure such as sets, linked lists, binary trees, rings, maps, graphs, etc.

The Ada subprograms specified in the following examples are organized according to the classification proposed by Parnas[12] (and used by Grady Booch[1]). Functions and procedures that make up a package interface fall into three categories:

- Selectors — operations which access, but do not modify an abstract data type.
- Constructors — operations which access and modify an abstract data type.
- Iterators — operations which sequentially enumerate the components of an abstract data type.

While annotating Ada package specifications, the *basic concept function*, which is often a selector but occasionally a constructor, may be used. These functions may be virtual or visible functions that are chosen not to be annotated. They serve as primitive functions upon which the annotations of other operations in the specification may be described. They serve as the basic concepts that are used to explain the semantics of the remaining operations.

## 4.1 List Package

```
-- Package Name:
--  List (generic)
-- Description:
--  A list is a sequence of zero or more items in which items can be added and removed from any position
-- such that a strict linear ordering is maintained.
-- Author:
--  John J. Kenney
-- Source:
--  Software Components With Ada, Grady Booch
--  List_Single_Unbounded_Unmanaged


   generic
      type Item is private;
   package List_Package is
      type List  is private;
      Null_List   : constant List;

      Overflow,
         -- This exception is raised if any of the routines below try to allocate more heap memory than
         -- available.
      Out_Of_Range : exception;
         -- This exception is raised if any of the routines below try to access a list item that does not
         -- exist.

   -- Basic Concept Functions

   -- The next two functions, Length_Of and Get_Item, are the basic concepts used in this package spec-
   -- ification. They are a complete observer basis for Lists.

      function Length_Of ( The_List : in List ) return Natural;
--|    where raise others => False;
         -- Return the current number of items in the list.

--:    function Get_Item ( The_List : in List;
--:                        Position  : in Positive ) return Item;
--|    where Position <= Length_Of( The_List );
         -- Return a copy of the Nth item in the list.
         -- This virtual function assumes that the given list has at least Position number of items.

   -- Selectors

      function Is_Null ( The_List : in List ) return Boolean;
--|    where raise others => False,
--|       return ( Length_Of( The_List ) = 0 );
         -- Return True if the list is a sequence of zero items.
```

```
        function Head_Of ( The_List : in List ) return Item;
--|      where Is_Null( The_List ) => raise Out_Of_Range,
--|            raise others => Is_Null( The_List ),
--|          return Get_Item( The_List, 1 );
            -- Return the first item from the sequence of items in a given list.

--:     function Equal ( Left, Right  : in List;
--:                      IndxL, IndxR: in Positive;
--:                      Len         : in Natural ) return Boolean;
--|      where ( Len > 0 ) -> ( IndxL+Len-1 <= Length_Of( Left ) and
--|                             IndxR+Len-1 <= Length_Of( Right )),
--|          return
--|              ( Len = 0 ) or else
--|              ( Get_Item( Left, IndxL ) = Get_Item( Right, IndxR ) and then
--|                Equal( Left, Right, IndxL+1, IndxR+1, Len-1 ));
            -- Return True if the two lists have the items in the same sequence for the given length.

        function Tail_Of ( The_List : in List ) return List;
--|      where Is_Null( The_List ) => raise Out_Of_Range,
--|            raise others => Is_Null( The_List ),
--|          return T : List =>
--|              Length_Of( T ) = Length_Of( The_List ) - 1 and then
--|              Equal( The_List, T, 2, 1, Length_Of( T ));
            -- Return the list denoting the tail of a given list.

        function Is_Equal ( Left, Right : in List ) return Boolean;
--|      where raise others => False,
--|          return Length_Of( Left ) = Length_Of( Right ) and then
--|                 ( Length_Of( Left) = 0 or else
--|                   ( Head_Of( Left ) = Head_Of( Right ) and
--|                     Is_Equal( Tail_Of( Left ), Tail_Of( Right ))));
            -- Return True if the two lists have the same state.

    -- Constructors

    -- Now, I will only use the Head_Of and Tail_Of functions to formally define the rest of the subpro-
    -- grams.

--:     function Append ( The_Item     : in Item;
--:                       To_The_List  : in List ) return List;
--|      where
--|          return New_List : List =>
--|              ( Head_Of( New_List ) = The_Item ) and
--|              ( Is_Equal( Tail_Of( New_List ), To_The_List ));
            -- Append an Item to the beginning of a list.
```

```
--:   function Append ( The_List     : in List;
--:                    To_The_List  : in List ) return List;
--|    where
--|       return
--|          if Is_Null( The_List ) then To_The_List
--|          else Append( Head_Of( The_List ),
--|                       Append( Tail_Of( The_List ), To_The_List ))
--|          end if;
         -- Append one list to the beginning of another list.

      procedure Construct ( The_Item      : in      Item;
                            And_The_List  : in out  List );
--|    where raise Overflow,
--|       out( Is_Equal( And_The_List, Append( The_Item, in And_The_List ))),
--|       out( Head_Of( And_The_List ) = The_Item ),
--|       out( Tail_Of( And_The_List ) = in And_The_List );
         -- Add an item to the head of a list.

      procedure Set_Head ( Of_The_List : in out List;
                           To_The_Item : in     Item );
--|    where Is_Null( Of_The_List ) => raise Out_Of_Range,
--|          raise others => Is_Null( Of_The_List ),
--|       out( Head_Of( Of_The_List ) = To_The_Item ),
--|       out( Tail_Of( Of_The_List ) = in Tail_Of( Of_The_List )),
--|       out( Is_Equal( Tail_Of( Of_The_List ), in Tail_Of( Of_The_List )));
         -- Set the value of the head of the list to the given item.

      procedure Copy ( From_The_List : in      List;
                       To_The_List   : in out  List );
--|    where raise Overflow,
--|       out( Is_Equal( From_The_List, To_The_List ));
         -- Copy the items from one list to another. Notice how this specification does not capture all
         -- the meaning of Copy.

      procedure Clear ( The_List : in out List );
--|    where raise others => False,
--|       out( Is_Null( The_List ));
         -- Remove all the items (if any) from the list and make the list null.

      procedure Swap_Tail ( Of_The_List  : in out List;
                            And_The_List : in out List);
--|    where Is_Null( Of_The_List ) => raise Out_Of_Range,
--|          raise others => Is_Null( Of_The_List ),
--|       out( Tail_Of( Of_The_List ) = in And_The_List ),
--|       out( Is_Equal( Tail_Of( Of_The_List ), in And_The_List )),
--|       out( And_The_List = in Tail_Of( Of_The_List ));
         -- Exchange the tail of one list with another list. We should never lose track of any part of
         -- either list.
```

```
    private
        type Node;
        type Node_Pointer is access Node;

--:     function No_Cycles ( The_List : in Node_Pointer ) return Boolean;
            -- Check that The_List does not have a cycle in it.

        type List is record
                Head : Node_Pointer := null;
            end record;
--|         where L : List => No_Cycles(L.Head);
            -- No list has a cycle in its list of nodes.

        Null_List   : constant List := List'(Head => null);
    end List_Package;

-- Commentary:

-- Implicit in this package specification is the concept structural sharing. Structural sharing occurs whenever
-- two or more names denote the same item or set of items. In general, it is dangerous to permit structural
-- sharing, since modification of an object via one name may have the unexpected side effect of altering the
-- object denoted by another name. For example, if A and B share the same list object, calling Clear with
-- the name A has the side effect of clearing the object that B denotes. B is thus placed into an inconsistent
-- state, as it now denotes a nonexistent object.

-- By declaring the type List as a private type, it is implicitly exporting the predefined operation of assign-
-- ment. The assignment operation has the same semantics as our structural sharing. But we also need the
-- semantics of Copy, where the items are duplicates, since a list is a polylithic component. Therefore, we
-- must provide both Copy and Share semantics and make sure that assignment always has Share seman-
-- tics, and also provide the copying semantics with the explicit constructor Copy. Thus, operations which
-- return lists may have two interpretations, one in which the items are duplicate copies, and another in
-- which the items are structurally shared.

-- We specify these concepts in Anna by using the predefined operation equality on lists for structural sharing
-- semantics and defining a virtual function Is_Equal for copy semantics. Notice that in our specifications
-- we mainly check that the heads of lists are structurally shared, while copies are verified recursively using
-- Is_Equal. The recursive function checks as a base case that the heads of the two lists have the same
-- value, and inductively (recursively) checks that the tails of the lists have the same values. Our definitions
-- are limited in that two structurally shared lists will also be Is_Equal, since their values are the same.

-- The private part of the specification is really a part of the implementation. Here we hint at what the
-- implementation of lists could be like. We can use Anna to specify important concepts very nicely, i.e.
-- that linked lists do not have cycles in them.
```

## 4.2 Deque Package

```
-- Package Name:
--  Deque (generic)
-- Description:
-- This package implements deques — bidirectional queues manipulated either at front or back.
-- Author:
--  Sriram Sankar
-- Source:
--  Software Components in Ada by Grady Booch


    generic
        type Item is private;
    package Deque is
        type Deque is limited private;
        type Location is ( Front, Back );

-- ———————— E X C E P T I O N S
        Overflow,
        Underflow : exception;

-- ———————— B A S I C   F U N C T I O N
        function Length ( D : Deque ) return Natural;
--|        where raise others => False;

-- ———————— S E L E C T O R S
        function Empty ( D : in Deque ) return Boolean;
--|        where return Length( D ) = 0;

        function Front_Of ( D : in Deque ) return Item;
--|        where Empty( D ) => raise Underflow;

        function Back_Of ( D : in Deque ) return Item;
--|        where Empty( D ) => raise Underflow;

--:      function Add ( I : Item; D : Deque; L : Location ) return Deque;
--|        where raise others => False,
--|              return Q : Deque =>
--|                  ( Length( Q ) = Length( D ) + 1 ) and
--|                  ( if Empty( D ) then
--|                          Front_Of( Q ) = Back_Of( Q ) and
--|                          Front_Of( Q ) = I
--|                    elsif L = Front then
--|                          Front_Of( Q ) = I
--|                    else
--|                          Back_Of( D ) = I
--|                    end if );

--:      function Pop ( D : Deque; L : Location ) return Deque;
--|        where raise others => False,
--|              return Q : Deque =>
--|                  Length( Q ) = Length( D ) - 1;
```

```
        function Equal ( Left, Right : in Deque ) return Boolean;
--|        where
--|             return ( if (Empty( Left ) or Empty( Right )) then
--|                                 Empty( Left ) and Empty( Right )
--|                        else
--|                                 Front_Of( Left ) = Front_Of( Right )  and
--|                                 Equal( Pop( Left, Front ), Pop( Right, Front ))
--|                        end if );

-- ---------- CONSTRUCTORS
        procedure Clear ( D : in out Deque );
--|        where raise others => False,
--|             out( Empty( D ));

        procedure Add ( I : Item; D : in out Deque; L : Location );
--|        where raise Overflow => D = in D,
--|             out( D = Add( I, in D, L ));

        procedure Pop ( D : in out Deque; L : Location );
--|        where Empty( D ) =>  raise Underflow,
--|               raise Underflow  =>  D = in D,
--|             out ( D = Pop( D, L ));

        procedure Copy ( From : in Deque; Into : in out Deque );
--|        where raise Overflow => Into = in Into,
--|             out ( Equal ( From, Into ) );

--| axiom
--|      for all D:Deque; I:Item =>
--|           Pop( Add( I, D, Front ), Front ) = D,
--|           Pop( Add( I, D, Back ), Back ) = D,
--|           if Empty( D ) then
--|                Pop( Add( I, D, Back ), Front ) = D
--|           else
--|                Pop( Add( I, D, Back ), Front ) =
--|                          Add( I, Pop( D, Front ), Back )
--|           end if,
--|           if Empty( D ) then
--|                Pop( Add( I, D, Front ), Back ) = D
--|           else
--|                Pop( Add( I, D, Front ), Back ) =
--|                          Add( I, Pop( D, Back ), Front )
--|           end if;

private
    type Deque_Header;
    type Deque is access Deque_Header;
end Deque;

-- Observations:
-- We have two interesting points: 1) the introduction of the virtual selectors, Add and Pop, and 2) the
-- recursive definition of the equality operator, Equal.
```

```
-- The virtual selectors, Add and Pop, are introduced because functions are cleaner to specify than pro-
-- cedures. Add and Pop are exported as procedures to reduce the copying of the limited private deque
-- type. The semantics of Add and Pop are defined to "remember" all of the previous Adds and Pops. An
-- Add or Pop function would have to return a new deque, which remembers all of the Adds and Pops of
-- the source deques, by allocating a large structure. An Add or Pop procedure is able to reuse the source
-- deque's memory, and does not have to allocate.

-- These virtual functions are not completely specified by their subprogram annotations. A complete
-- subprogram annotation would have to explicitly iterate through the deque type, limiting the method by
-- which the deque may remember its previous history. Completing the specification by using axioms is an
-- additional way of specifying this history.

-- The recursive definition of the equality operator, Equal, uses the functions Empty and Front as the base
-- cases and the virtual function Pop as the inductive step. It tests that either both deques have nothing,
-- or they have the same items at the front and after popping off those front items, the rest of the deques
-- are also the same.
```

## 4.3 Tree Package

```
-- Package Name:
--  Tree (generic)
-- Description:
--  This package manipulates an arbitrary tree. The specification combines the notion of a node and a tree;
-- a node is viewed as being the root of the subtree. Thus, the formal specification has no node type, but it
-- is implicitly available in the Tree type.
--  A tree is a collection of nodes that can have an arbitrary number of references to other nodes; there
-- can be no cycles or short-circuit references, and for every two nodes there exists a unique simple path
-- connecting them. Each node can have an arbitrary number of children.
-- Author:
--  John Kenney
-- Source:
--  Software Components in Ada by Grady Booch

    generic
        type Item is private;
    package   Tree_Package is
        type Tree         is limited private;
--:     type Tree_List    is limited private;
            -- A tree_list would preferably be a subtype of a list type derived from the instantiation of a
            -- list package (with Tree as its formal generic parameter). However, Ada rules don't allow
            -- such a parameter to be an incomplete type.

        Overflow      : exception;
        Tree_Is_Null  : exception;


-- Basic Concept Functions

--:    function The_Heap_Is_Exhausted return Boolean;

--:    function Length_Of ( T : in Tree_List ) return Natural;

--:    function Get_Item ( T : in Tree_List; N : in Positive ) return Tree;
--|     where 1 <= N <= Length_Of(T);

        function Is_Null ( T : in Tree ) return Boolean;
--|     where raise others => False;

        function Value_Of ( T : in Tree ) return Item;
--|     where Is_Null(T)       => raise Tree_Is_Null,
--|           raise others     => Is_Null(T);

        function Parent_Of ( T : in Tree ) return Tree;
--|     where Is_Null(T)       => raise Tree_Is_Null,
--|           raise others     => Is_Null(T);
```

```
-- Selectors

--:   function Is_Member ( T : in Tree; S : in Tree_List ) return Boolean;
--|    where raise others => False,
--|       return exist N : 1 .. Length_Of(S) => Get_Item(S, N) = T;

--:   function Children_Of ( T : in Tree ) return Tree_List;
--|    where raise others => False,
--|       return S : Tree_List =>
--|          ( for all D : Tree =>
--|                Is_Member(D, S) <-> ( Parent_Of(D) = T or Is_Null(D)));

      function Degree ( T : in Tree ) return Natural;
--|    where return Length_Of( Children_Of(T));

      function Is_Leaf ( T : in Tree ) return Boolean;
--|    where return Length_Of( Children_Of(T)) = 0;

      function Is_Interior ( T : in Tree ) return Boolean;
--|    where return Length_Of( Children_Of(T)) /= 0;

      function Child_Of ( T : in Tree; N : in Positive ) return Tree;
--|    where Degree(T) < N => raise Tree_Is_Null,
--|       return Get_Item( Children_Of(T), N);

      function "=" ( Left, Right : in Tree ) return Boolean;
--|    where raise others => False,
--|       return if ( Is_Null(Left) or Is_Null(Right)) then
--|                    Is_Null(Left) and Is_Null(Right)
--|              else Value_Of(Left) = Value_Of(Right) and then
--|                    Degree(Left) = Degree(Right) and then
--|                  ( for all N : Natural range 1 .. Degree(Left) =>
--|                        Child_Of(Left, N) = Child_Of(Right, N))
--|              end if;

      function Is_Ancestor ( T1, T2 : in Tree ) return Boolean;
--|    where
--|       return ( Parent_Of(T2) = T1 ) or else
--|              ( exist T3 : Tree => Parent_Of(T2) = T3 and Is_Ancestor(T1, T3));

      function Is_Descendent ( T1, T2 : in Tree ) return Boolean;
--|    where return Is_Ancestor(T2, T1);

-- Constructors
      procedure Clear ( T : in out Tree );
--|    where raise others => False,
--|       out( Is_Null(T));
```

```
      procedure Construct ( The_Item              : in      Item;
                            And_The_Tree          : in out  Tree;
                            Number_Of_Children    : in      Natural;
                            On_The_Child          : in      Natural);
--|   where The_Heap_Is_Exhausted    => raise Overflow,
--|         raise others             => The_Heap_Is_Exhausted,
--|      out( Value_Of( And_The_Tree ) = The_Item ),
--|      out( Degree( And_The_Tree ) = Number_Of_Children ),
--|      out( for all N : Natural range 1 .. Number_Of_Children =>
--|                if ( N = On_The_Child ) then
--|                      Child_Of( And_The_Tree, N ) = in And_The_Tree
--|                else Is_Null( Child_Of( And_The_Tree, N ))
--|                end if );
        -- Add an item at the root of the tree, created with the given degree; the original tree becomes
        -- the given child of the new node. Number_Of_Children specifies the number of children to
        -- be created. Construct must permit the creation of nodes with no children.

      procedure Set_Item (T : in out Tree; I : in Item);
--|    where raise others => False,
--|      out( Value_Of (T) = I),
--|      out( Degree(T) = in( Degree(T) )),
--|      out( Children_Of(T) = in( Children_Of(T) ));
        -- Set the value of the root of the tree to the given item.

      procedure Swap_Child ( The_Child      : in      Positive;
                             Of_The_Tree    : in out  Tree;
                             And_The_Tree   : in out  Tree);
--|    where Degree( Of_The_Tree ) < The_Child => raise Tree_Is_Null,
--|          Degree( Of_The_Tree ) >= The_Child and
--|          The_Heap_Is_Exhausted => raise Overflow,
--|          raise others => Degree( Of_The_Tree ) < The_Child or
--|                          The_Heap_Is_Exhausted,
--|       out( Child_Of( Of_The_Tree, The_Child ) = in And_The_Tree ),
--|       out( And_The_Tree = Child_Of( in Of_The_Tree, The_Child )),
--|       out( for all N : Positive range 1 .. Degree( Of_The_Tree ) =>
--|                N /= The_Child -> Child_Of( Of_The_Tree, N )
--|                                  = Child_Of( in Of_The_Tree, N ));
        -- Exchange the given child of one tree with another entire tree.

      procedure Copy ( From : in Tree; Tx : in out Tree );
--|    where raise Overflow => not The_Heap_Is_Exhausted,
--|       out( Tx = From );

--|axiom
--|    for all S : Tree_Package'Type; T1, T2 : Tree; N : Positive =>
--|       S [ Copy(T1, T2); Clear(T2)].Is_Null(T1) -> Is_Null(T1),
--|       S [ Swap_Child(N, T1, T2); Clear(T2)].
--|             Is_Null( Child_Of(T1, N)) -> Is_Null(T2),
--|       S [ Swap_Child(N, T1, T2); Clear(T2)].
--|             Is_Null( Child_Of(T1, N)) -> Is_Null(T2);
```

```
    private
        type Tree_Rec;
        type Tree is access Tree_Rec;
        Null_Tree : constant Tree := null;

--:     type Tree_List_Rec;
--:     type Tree_List is access Tree_List_Rec;
    end Tree_Package;

-- Commentary:

-- This package specifies the behavior of arbitrary trees. A reader who only has an informal description may
-- read an Ada specification and be unsure if leaf nodes of arbitrary trees may be null. This specification
-- specifies in the constructors that they may.

-- Successor Package State annotations are used in the axioms to specify completely the semantics of copy.
-- The axioms specify the results of applying sequences of operations on trees (e.g. the result of a Copy
-- followed by a Clear). The Copy procedure is intended to make a new copy of the given tree; that is, the
-- two trees do not become aliases for the same underlying structure, but a duplicate physical representation
-- is generated. The "no sharing" axiom requires (using successor package states) that when we copy T1
-- to T2 and then make T2 null, the only way that T1 can be null is if T2 was originally null.
```

## 4.4 Graph Package

```
-- Package Name:
-- Graph (generic)
-- Description:
-- This is a generic graph package. A graph is a collection that includes a set of vertices and a set of
-- arcs. A vertex, aka a node, forms the basic structural element of the graph. An arc, aka an edge, is a
-- connection between two vertices.
-- The graph is directed, where the order of endpoints of an arc is important. The graph is also labeled,
-- where each vertex and arc has an associated item or attribute. Thus, arcs and vertices have values (items).
-- Author:
-- Walter Mann, John Kenney, and Rob Chang
-- Source:
-- Software Components in Ada by Grady Booch

   generic
      type Item        is private; -- the value of a vertex.
      type Attribute   is private; -- the value of an arc.
   package Graph_Package is

      type Graph is limited private;
         -- a collection that includes a set of vertices and a set of arcs.
      type Vertex is private;
         -- aka a node, forms the basic structural element of the graph.
      type Arc is private;
         -- aka an edge, is a connection between two vertices.

      Null_Vertex : constant Vertex;
      Null_Arc    : constant Arc;

-- ---------- EXCEPTIONS

      Overflow                : exception;
      Vertex_Is_Null          : exception;
      Vertex_Is_Not_In_Graph  : exception;
      Vertex_Has_References   : exception;
      Arc_Is_Null             : exception;
      Arc_Is_Not_In_Graph     : exception;

-- ---------- BASIC  FUNCTIONS

      function Number_Of_Vertices_In ( The_Graph : Graph ) return Natural;

      function Number_Of_Arcs_In ( The_Graph : Graph ) return Natural;

      function Number_Of_Arcs_From ( The_Vertex : Vertex ) return Natural;

--:    function Number_Of_Arcs_To ( The_Vertex : Vertex ) return Natural;

      function Is_A_Member ( The_Vertex    : Vertex;
                             Of_The_Graph : Graph ) return Boolean;
```

```
        function Is_A_Member ( The_Arc        : Arc;
                               Of_The_Graph   : Graph ) return Boolean;

        function "="( Graph1, Graph2 : Graph ) return Boolean;

-- ---------- SELECTORS

        function Is_Empty ( The_Graph : Graph ) return Boolean;
--|      where
--|          return Number_Of_Vertices_In( The_Graph ) = 0;

        function Is_Null ( The_Vertex : Vertex ) return Boolean;
--|      where
--|          return The_Vertex = Null_Vertex;
           -- Return true if the object does not denote any vertex.

        function Is_Null ( The_Arc : Arc ) return Boolean;
--|      where
--|          return The_Arc = Null_Arc;
           -- Return true if the object does not denote any arc.

        function Item_Of ( The_Vertex : Vertex ) return Item;
--|      where Is_Null( The_Vertex ) => raise Vertex_Is_Null;
           -- Return the item from the designated vertex.

        function Attribute_Of ( The_Arc : Arc ) return Attribute;
--|      where Is_Null( The_Arc ) => raise Arc_Is_Null;
           -- Return the attribute from the designated arc.

        function Source_Of ( The_Arc : Arc ) return Vertex;
--|      where Is_Null( The_Arc ) => raise Arc_Is_Null;

        function Destination_Of ( The_Arc : Arc ) return Vertex;
--|      where Is_Null( The_Arc ) => raise Arc_Is_Null;

-- ---------- CONSTRUCTORS

        procedure Clear ( The_Graph : in out Graph );
--|      where
--|          out( Number_Of_Vertices_In( The_Graph ) = 0 );

        procedure Add ( The_Vertex     : in out Vertex;
                        With_The_Item  : in     Item;
                        To_The_Graph   : in out Graph);
--|      where raise Overflow,
--|          out( Number_Of_Vertices_In( To_The_Graph )
--|                   = in Number_Of_Vertices_In( To_The_Graph ) + 1 ),
--|          out( Is_A_Member( The_Vertex, To_The_Graph )),
--|          out( Item_Of( The_Vertex ) = With_The_Item );
```

```
        procedure Remove ( The_Vertex       : in out Vertex;
                           From_The_Graph : in out Graph);
--|     where Is_Null( The_Vertex ) => raise Vertex_Is_Null,
--|           Number_Of_Arcs_To( The_Vertex ) /= 0 => raise Vertex_Has_References,
--|           not Is_A_Member( The_Vertex, From_The_Graph )
--|                                => raise Vertex_Is_Not_In_Graph,
--|       out( Number_Of_Arcs_From( in The_Vertex ) = 0 ),
--|       out( not Is_A_Member( in The_Vertex, From_The_Graph )),
--|       out( Is_Null( The_Vertex )),
--|       out( Number_Of_Vertices_In( From_The_Graph )
--|                = in Number_Of_Vertices_In( From_The_Graph ) - 1 );
          -- Destroy the designated vertex in the graph.

        procedure Set_Item ( Of_The_Vertex : in out Vertex;
                             To_The_Item   : in      Item );
--|     where Is_Null( Of_The_Vertex ) => raise Vertex_Is_Null,
--|       out( Item_Of( Of_The_Vertex ) = To_The_Item );
          -- Set the value of the designated vertex to the given item.

        procedure Create ( The_Arc              : in out Arc;
                           With_The_Attribute   : in      Attribute;
                           From_The_Vertex      : in out Vertex;
                           To_The_Vertex        : in      Vertex;
                           In_The_Graph         : in out Graph );
--|     where Is_Null( From_The_Vertex ) or Is_Null( To_The_Vertex )
--|                                => raise Vertex_Is_Null,
--|           not Is_A_Member( From_The_Vertex, In_The_Graph ) or
--|           not Is_A_Member( To_The_Vertex, In_The_Graph )
--|                                => raise Vertex_Is_Not_In_Graph,
--|       out( Attribute_Of( The_Arc ) = With_The_Attribute ),
--|       out( Source_Of( The_Arc ) = From_The_Vertex ),
--|       out( Destination_Of( The_Arc ) = To_The_Vertex ),
--|       out( Number_Of_Arcs_To ( To_The_Vertex )
--|                = Number_Of_Arcs_To ( To_The_Vertex ) + 1 ),
--|       out( Number_Of_Arcs_From( From_The_Vertex )
--|                = in Number_Of_Arcs_From( From_The_Vertex ) + 1 );

        procedure Destroy ( The_Arc        : in out Arc;
                            In_The_Graph   : in out Graph );
--|     where Is_Null( The_Arc ) => raise Arc_Is_Null,
--|           not Is_A_Member( The_Arc, In_The_Graph )
--|                           => raise Arc_Is_Not_In_Graph,
--|       out( Number_Of_Arcs_In( In_The_Graph )
--|                = in Number_Of_Arcs_In( In_The_Graph ) - 1 ),
--|       out( not Is_A_Member( in The_Arc, In_The_Graph )),
--|       out( The_Arc = Null_Arc ),
--|       out( Number_Of_Arcs_To( in Destination_Of( The_Arc ))
--|                = in Number_Of_Arcs_To( Destination_Of( The_Arc )) - 1 );
          -- Remove the given arc in the graph.
```

```
        procedure Set_Attribute ( Of_the_Arc          : in out Arc;
                                  To_The_Attribute    : in     Attribute );
--|      where Is_Null( Of_The_Arc ) => raise Arc_Is_Null,
--|          out( Attribute_Of( Of_the_Arc ) = To_The_Attribute );
            -- Set the value of the designated arc to the given name.

        procedure Copy ( From_The_Graph : in     Graph;
                         To_The_Graph   : in out Graph );
--|      where
--|          out( From_The_Graph = To_The_Graph );

        generic
            with procedure Process ( V : in Vertex; Continue : out Boolean );
        procedure Iterate_Vertices ( Over_The_Graph : in Graph );

        generic
            with procedure Process ( A : in Arc; Continue : out Boolean );
        procedure Iterate_Arcs ( Over_The_Graph : in Graph );

        generic
            with procedure Process ( A : in Arc; Continue : out Boolean );
        procedure Reiterate ( Over_The_Vertex : in Vertex );

-- ———————— A X I O M S

--|axiom
--|    Number_Of_Vertices_In ( Graph'Initial ) = 0,
--|    Number_Of_Arcs_In ( Graph'Initial ) = 0,
--|    Number_Of_Arcs_From ( Vertex'Initial ) = 0,
--|    Number_Of_Arcs_To ( Vertex'Initial ) = 0;
--|    for all V : Vertex => not Is_A_Member ( V, Graph'Initial ),
--|    for all A : Arc => not Is_A_Member ( A, Graph'Initial );

private

    type Vertex_Node;
    type Vertex is access Vertex_Node;
    Null_Vertex : constant Vertex := null;

    type Arc_Node;
    type Arc is access Arc_Node;
    Null_Arc : constant Arc := null;

    type Graph_Node;
    type Graph is access Graph_Node;

end Graph_Package;
```

```
-- Commentary:
-- In subprograms, it is often the case that a few properties of the state change, but many do not. For
-- example, in subprogram Add, when adding a vertex to a graph, the only changes to values of functions
-- on the graph are that the number of vertices in the graph is incremented, and the given vertex becomes a
-- member of the graph. All other vertices and arcs in the graph remain unchanged. Rather than specifying
-- all properties of the state which do not change, the designer can apply a "frame axiom," which states
-- that whatever has not been explicitly expressed to change remains unchanged. However, it is common
-- to express other information about invariance in the package axioms.

-- This package provides an equality operator on the type Graph, which is used in annotating subprogram
-- Copy. It is not further specified, but by Anna rules it must obey the implicit equality axioms: reflexivity,
-- symmetry, transitivity, substitution for functions, and independence of the package state. These axioms
-- are adequate to insure that two "equal" Graphs behave identically in terms of visible functions of the
-- package.

-- One advantage of a formal specification is that it is often easy to extend the package with other subpro-
-- grams which operate on graphs; the extensions have straightforward specifications in terms of the basic
-- concepts already defined. For example, a new Boolean function Path, which returns True if there is a
-- path of arcs between two vertices, is easy to specify in Anna:

   function Path( From_Vertex, To_Vertex : in Vertex ) return Boolean;
--| where
--|    return From_Vertex = To_Vertex or else
--|           ( exist V : Vertex; A : Arc =>
--|                  Source_Of(A) = From_Vertex and
--|                  Destination_Of(A) = V and
--|                  Path(V,To_Vertex) );
```

## 4.5 Map Package

```
-- Description
--  A map is a function on elements of one type, called the domain, yielding elements of a second type, called
-- range. In other words, a map m from Domain_Set to Range_Set exists if for each x in Domain_Set we
-- can specify a unique element in Range_Set, which I denote m*x. It is helpful to think of m as a rule
-- which assigns to each element x in Domain_Set a unique element m*x in Range_Set. The element m*x
-- is usually called the value of m at x. The important points are that the map m is defined if for every x in
-- Domain_Set there exists a m*x, and that there is just one such element for each x.
--  Formally, we can view the value of a map as an unordered collection of ordered pairs consisting of an
-- element of the domain and an element of the range. The domain and the range are typically different
-- types, although they may be the same type. Each ordered pair thus denotes the binding of two elements.
-- For every element of the domain, there can exist no more than one element of the range. The inverse
-- does not hold; every element of the range can be associated with zero or more elements of the domain.
-- Author:
--  David S. Rosenblum, Rob Chang, and John Kenney.
-- Source:
--  Software Components in Ada by Grady Booch.

  generic
     type Domain is private;
     type Ranges is private;
     Number_Of_Buckets : in Positive;
     with function Hash_Of ( The_Domain : in Domain ) return Positive;
--|  for all D : Domain => Hash_Of( D ) <= Number_Of_Buckets;
  package Map_Package is
     type Map   is limited private;
--:  type Pair   is limited private;

     Domain_Is_Not_Bound,
          -- There does not currently exist a binding for the given element of the domain.
     Multiple_Binding,
          -- There already exists a binding for the given element of the domain.
     Overflow : exception;
          -- The map cannot grow large enough to complete the desired operation.

--:   function Domain_Of ( The_Pair : in Pair ) return Domain;
--|   where raise others => False;

--:   function Ranges_Of ( The_Pair : in Pair ) return Ranges;
--|   where raise others => False;

     function Extent_Of ( The_Map : in Map ) return Natural;
--|   where raise others => False;
        -- Return the current number of ordered pairs in the map.

     function Is_Empty ( The_Map : Map ) return Boolean;
--|   where raise others => False,
--|        return Extent_Of( The_Map ) = 0;
        -- Return True if the map contains no ordered pairs.
```

```
      procedure Clear ( The_Map : in out Map );
--|   where raise others => False,
--|       out( Is_Empty( The_Map ));
          -- Remove all the ordered pairs (if any) from the map and make the map empty.

--:   function A_Pair_Of ( The_Map : in Map ) return Pair;
--|   where not Is_Empty( The_Map ),
--|         raise others => False;

--:   function "-" ( The_Domain : in Domain;
--:                  In_The_Map : in Map ) return Map;
--|   where raise others => False,
--|       return New_Map : Map =>
--|           not exist Any_Pair : Pair =>
--|                     ( Any_Pair = A_Pair_Of( New_Map )) and
--|                     ( Domain_Of( Any_Pair ) = The_Domain );

      function Is_Bound ( The_Domain : in Domain;
                          In_The_Map : in Map ) return Boolean;
--|   where raise others => False,
--|       return ( not Is_Empty( In_The_Map )) and then
--|              ( The_Domain = Domain_Of( A_Pair_Of( In_The_Map )) or else
--|                Is_Bound( The_Domain,
--|                          "-"( Domain_Of( A_Pair_Of( In_The_Map )), In_The_Map )));
          -- Return True if there is an element of the range corresponding to the given element of the
          -- domain in the map.

      function Defined ( The_Map : in Map ) return Boolean;
--|   where raise others => False,
--|       return for all The_Domain : Domain => Is_Bound(The_Domain, The_Map);

--:   function "*" ( In_The_Map : in Map;
--:                  The_Domain : in Domain ) return Ranges;
--|   where not Is_Bound( The_Domain, In_The_Map ) => raise Domain_Is_Not_Bound,
--|         raise others => not Is_Bound( The_Domain, In_The_Map ),
--|       return
--|           if ( The_Domain = Domain_Of( A_Pair_Of( In_The_Map ))) then
--|              Ranges_Of( A_Pair_Of( In_The_Map ))
--|           else
--|              "*"("-"( Domain_Of( A_Pair_Of( In_The_Map )), In_The_Map ), The_Domain )
--|           end if;

      function Range_Of ( The_Domain : in Domain;
                          In_The_Map : in Map ) return Ranges;
--|   where not Is_Bound( The_Domain, In_The_Map )
--|                                    => raise Domain_Is_Not_Bound,
--|         raise others => not Is_Bound( The_Domain, In_The_Map ),
--|       return In_The_Map * The_Domain;
          -- Return an element of the range corresponding to the given element of the domain in the
          -- map.
```

```
    procedure Bind ( The_Domain     : in      Domain;
                     And_The_Range  : in      Ranges;
                     In_The_Map     : in out  Map );
--| where Is_Bound( The_Domain, In_The_Map ) => raise Multiple_Binding,
--|       raise Overflow,
--|    out( Is_Bound( The_Domain, In_The_Map )),
--|    out( And_The_Range = In_The_Map * The_Domain );
      -- Add an ordered pair consisting of an element of the domain and an element of the range to
      -- the map.

    procedure Unbind ( The_Domain : in      Domain;
                       In_The_Map : in out  Map );
--| where not Is_Bound( The_Domain, In_The_Map ) => raise Domain_Is_Not_Bound,
--|       raise others => not Is_Bound( The_Domain, In_The_Map ),
--|    out( not Is_Bound( The_Domain, In_The_Map ));
      -- Remove the ordered pair for a given element of the domain from the map.

--: function "<=" ( Left, Right : in Map ) return Boolean;
--| where raise others => False,
--|    return
--|       if Is_Empty( Left ) then True
--|       else
--|          Is_Bound( Domain_Of( A_Pair_Of( Left )), Right ) and then
--|          ( Left * Domain_Of( A_Pair_Of( Left ))
--|                = Right * Domain_Of( A_Pair_Of( Left ))) and then
--|          "<="( "-"( Domain_Of( A_Pair_Of( Left )), Left ), Right )
--|       end if;

    function "=" ( Left, Right : in Map ) return Boolean;
--| where raise others => False,
--|    return ( Left <= Right ) and then ( Right <= Left );

    procedure Copy ( From_The_Map : in      Map;
                     To_The_Map   : in out  Map );
--| where raise Overflow,
--|    out( To_The_Map = From_The_Map );
      -- Copy the ordered pairs from one map to another map.

    function  Is_Surjection ( The_Map : Map ) return Boolean;
--| where
--|    return
--|       for all R : Ranges =>
--|          exist D : Domain => The_Map * D = R;
      -- The map from Domain to Ranges is a surjection if every R in Ranges is a value The_Map*D
      -- for at least one D in the Domain.

    function  Is_Injection ( The_Map : Map ) return Boolean;
--| where
--|    return
--|       for all D1, D2 : Domain =>
--|          D1 /= D2 -> The_Map * D1 /= The_Map * D2;
      -- The map from Domain to Ranges is an injection if every R in Ranges is a value The_Map*D
      -- for at most one D in the Domain.
```

```
      function   Is_Bijection ( The_Map : Map ) return Boolean;
--|    where
--|        return Is_Surjection( The_Map ) and Is_Injection( The_Map );
           -- The map from Domain to Ranges is a bijection if it is both a surjection and an injection,
           -- that is, if every R in Ranges is a value The_Map*D for exactly one D in the Domain.

--|axiom
--|    Extent_Of( Map'Initial ) = 0,
--|    for all D : Domain => not Is_Bound( D, Map'Initial );

   private
      type Map_Rec;
      type Map is access Map_Rec;
--:   type Pair_Rec;
--:   type Pair is access Pair_Rec;
   end Map_Package;

-- Commentary:
-- This package uses the virtual function A_Pair_Of to refer to an arbitrary Pair element of a Map. In this
-- way, later functions can be defined by recursively examining and removing each element of the map.
```

## 4.6 Rings Package

```
-- Package Name:
--  Rings (generic)
-- Description:
--  This package provides Ring manipulation procedures such as marking items in a ring, insertion, and
-- deletion of items, copying of Rings and rotation. A ring is a circular buffer where the current "top" can
-- rotate in either direction.
-- Author:
--  Randall Neff
-- Source:
--  Software Components in Ada by Grady Booch

generic
    type Item is private;

package Rings is
    type Ring is limited private;
        -- a sequence of zero or more items arranged in a circular fashion.
    type Direction is ( Forward, Backward );

-- ---------- E X C E P T I O N S
    Overflow : exception;
        -- The ring cannot grow large enough to complete the desired operation.
    Underflow : exception;
        -- The ring is already empty.
    Rotate_Error : exception;
        -- There is nothing in the ring to rotate.

-- ---------- B A S I C   F U N C T I O N S
--: function Mark_Is ( The_Ring : in Ring ) return Natural;
--| where raise others => False;
        -- returns the value of the Mark as an index to the Ring

     function Extent_Of ( The_Ring : in Ring ) return Natural;
--| where raise others => False;
        -- returns the current number of Items in a ring.

--: function Index ( The_Ring : in Ring; I : in Positive ) return Item;
--| where I <= Extent_Of( The_Ring ),
--|           raise others => False;
        -- returns the I'th Item stored in the ring.

-- ---------- S E L E C T O R S
     function Is_Empty ( The_Ring : in Ring ) return Boolean;
--| where raise others => False,
--|     return Extent_Of( The_Ring ) = 0;
```

```
    function Is_Equal ( Left, Right : in Ring ) return Boolean;
--| where raise others => False,
--|    return Extent_Of( Left ) = Extent_Of( Right ) and then
--|           Mark_Is( Left ) = Mark_Is( Right ) and then
--|           ( for all I : 1 .. Extent_Of( Left ) =>
--|                   Index( Left, I ) = Index( Right, I ) );

    function Top_Of ( The_Ring : in Ring ) return Item;
--| where
--|    Is_Empty( The_Ring ) => raise Underflow,
--|    return Index( The_Ring, 1 );

    function At_Mark ( The_Ring : in Ring ) return Boolean;
--| where raise others => False,
--|    return if Extent_Of( The_Ring ) = 0 then
--|              Mark_Is( The_Ring ) = 0
--|           else
--|              Mark_Is( The_Ring ) = 1
--|           end if;

-- ———————— CONSTRUCTORS
    procedure Copy ( From_The_Ring : in      Ring;
                     To_The_Ring   : in out Ring );
--| where raise Overflow,
--|    out( Is_Equal( From_The_Ring, To_The_Ring ) );

    procedure Clear ( The_Ring : in out Ring );
--| where raise others => False,
--|    out( Is_Empty( The_Ring ) );

    procedure Insert ( The_Item : Item; In_The_Ring : in out Ring );
--| where raise Overflow,
--|    out( Extent_Of( In_The_Ring ) = Extent_Of( in In_The_Ring) +1 ),
--|    out( Top_Of( In_The_Ring ) = The_Item ),
--|    out( Mark_Is( In_The_Ring ) = Mark_Is( in In_The_Ring ) + 1 ),
--|    out( for all I : 2 .. Extent_Of( In_The_Ring) =>
--|              Index( In_The_Ring, I ) = Index( in In_The_Ring, I-1) );

    procedure Pop ( The_Ring : in out Ring );
--| where Is_Empty( The_Ring ) => raise Underflow,
--|    out( Extent_Of( The_Ring ) = Extent_Of( in The_Ring) -1 ),
--|    out( if Mark_Is( in The_Ring ) = 1 then
--|            Mark_Is( The_Ring ) =  1
--|         else
--|            Mark_Is( The_Ring) = Mark_Is( in The_Ring ) -1
--|         end if ),
--|    out( for all I : 1 .. Extent_Of( The_Ring) =>
--|              Index( The_Ring, I ) = Index( in The_Ring, I+ 1) );
```

```
--: function Shift ( Top       : in Positive;
--:                  Distance  : in Integer;
--:                  Extent    : in Positive ) return Positive;
--| where raise others => False,
--|    return
--|        if ( Top - Distance ) mod Extent = 0 then
--|             Extent
--|        else
--|             ( Top - Distance ) mod Extent
--|        end if;

--: function Rotate ( The_Ring          : in Ring;
--:                   Distance          : in Integer ) return Ring;
--| where not Is_Empty( The_Ring ),
--|    return New_Ring : Ring =>
--|      ( Extent_Of( New_Ring ) = Extent_Of( The_Ring )) and then
--|      ( for all I : 1 .. Extent_Of( The_Ring) =>
--|           Index( New_Ring, I )
--|             = Index( The_Ring, Shift( I, Distance, Extent_Of( The_Ring ))))
--|      and then
--|      ( Mark_Is( New_Ring )
--|            = Shift( Mark_Is( The_Ring ), Distance, Extent_Of( The_Ring )));

   procedure Rotate ( The_Ring          : in out Ring;
                      In_The_Direction  : in     Direction);
--| where Is_Empty( The_Ring ) => raise Rotate_Error,
--|    out( The_Ring = if In_The_Direction = Forward then
--|                          Rotate( in The_Ring, 1 )
--|                    else
--|                          Rotate( in The_Ring, -1 )
--|                    end if );

   procedure Mark ( The_Ring : in out Ring );
--| where raise others => False,
--|    out( if Is_Empty( The_Ring ) then
--|                 Mark_Is( The_Ring ) = 0
--|         else
--|                 Mark_Is( The_Ring ) = 1
--|         end if );

   procedure Rotate_To_Mark (The_Ring : in out Ring);
--| where raise others => False,
--|    out( The_Ring = Rotate( in The_Ring, in Mark_Is( The_Ring ) - 1 ));

-- --------ITERATORS
   generic
      with procedure Process ( The_Item : in      Item;
                               Continue :    out Boolean);
   procedure Iterate ( Over_The_Ring : Ring );
```

```
-- ———————— A X I O M S
--| axiom
--|     for all Pack : Rings'Type;
--|             R1, R2   : Ring;
--|             I1, I2   : Item =>
--|         Pack [ Insert( I1, R1 ); Pop( R1 ) ] = Pack,
--|         Pack [ Rotate( R1, Forward ); Rotate( R1, Backward ) ] = Pack,
--|         Pack [ Rotate( R1, Backward ); Rotate( R1, Forward ) ] = Pack,
--|         Pack [ Mark( R1 ) ].At_Mark( R1 ),
--|         Pack [ Rotate_To_Mark( R1 ) ].At_Mark( R1 ),
--|         Pack [ Insert( I1, R1 ); Mark( R1 ); Rotate( R1, Forward );
--|                    Rotate_To_Mark( R1 ) ].Top_Of( R1 ) = I1,
--|         Pack [ Insert( I1, R1 ); Mark( R1 ); Rotate( R1, Backward );
--|                    Rotate_To_Mark( R1 ) ].Top_Of( R1 ) = I1;

private
    type Node;
    type Structure is access Node;
    type Ring is record
                    The_Top   : Structure;
                    The_Mark  : Structure;
                end record;
end Rings;

-- Commentary:
-- One feature of this package specification is the use of the virtual Index function. It abstracts the Ring
-- as an array that can be indexed. Individual elements can be read from any part of the ring within the
-- annotations. Elements are numbered starting at 1; the index 0 is returned for empty Rings. Similarly,
-- the virtual function Mark_Is returns the index number of the element that is currently 'marked' within
-- the Ring. The index 0 is used for the location of the Mark in an empty Ring.

-- We also define a general, virtual concept of rotating rings. Using the virtual Rotate function, rings can
-- be rotated by any integer value (where positive numbers correspond to forward shift, and negative to
-- backward shift). The virtual function Shift captures the algebraic expression denoting shifting index
-- values modulo the ring size. Then the visible Rotate and Rotate_To_Mark procedures are simple to
-- specify.

-- Many of the axioms given are not strictly necessary. For instance, the axiom
--                     Pack [ Mark( R1 ) ].At_Mark( R1 )
-- can be logically deduced from the subprogram annotations for Mark and At_Mark. However, this kind
-- of redundancy is very useful as an extra check that the designer understands the implications of the
-- specification.
```

# Chapter 5

# Other Examples

This chapter contains some simple package specifications that are not definitions of abstract data types. This chapter shows Anna being used to annotate other kinds of packages, including packages that use Floating point numbers. The examples are:

- A Bank Automatic Teller Machine
- Math Functions Package
- Complex Number Package
- Generic Sort Package

## 5.1 Bank ATM Packages

```
-- Package Name:
--  ATM_Utilities
-- Description:
--  Encapsulates types needed to support an Automatic Teller machine system.
-- Author:
--  John Kenney and Walter Mann


package ATM_Utilities is

    type Dollars_Type is delta 0.001 range -1_000_000.0 .. 1_000_000.0;

    subtype Positive_Dollars_Type is
                    Dollars_Type range 0.0 .. Dollars_Type'Last;

    type Check_Status_Type is (Not_Yet_Received, Being_Verified, Stopped, Cleared);

    type Account_Type is (Checking, Savings, Loan);

    type Check_Number_Type is new Positive;

    type Account_Array is array (Account_Type) of Positive_Dollars_Type;

    type Check_Status_Array is array (Check_Number_Type range <>)
                                        of Check_Status_Type;

    type Check_Amount_Array is array (Check_Number_Type range <>)
                                        of Positive_Dollars_Type;

    type Account_State( Max_Check : Check_Number_Type ) is record
                    -- Max_Check is the highest numbered check that has been issued to the customer.
                Acct_Balances : Account_Array;
                    -- Balances of accounts at start of ATM session.
                Check_Status : Check_Status_Array( 1 .. Max_Check );
                    -- Status of all known checks at start of ATM session.
                Check_Amount : Check_Amount_Array( 1 .. Max_Check );
                    -- Amount of all received checks at start of ATM session.
                Line_Of_Credit : Positive_Dollars_Type;
                    -- Maximum amount a customer can borrow from his Loan account.
        end record;
--|      where State : Account_State =>
--|                  State.Acct_Balances(Loan) <= State.Line_Of_Credit;
    -- This type represents the state of a customer's accounts; it should describe completely the account
    -- information which a customer has access to.

end ATM_Utilities;
```

```
-- Package Name:
--  ATM_Session
-- Description:
--  This generic package operates as a "shell" invoked when a customer has correctly inserted his card and
-- given his code. It provides a user-interface of options to manipulate a single account. The account infor-
-- mation is derived from the generic formal parameter Acct_State; presumably this package is instantiated
-- with that customer's account information, and the updated state will be saved when the session terminates.
-- Author:
--  John Kenney, Walter Mann, and Doug Bryan


with ATM_Utilities; use ATM_Utilities;
generic
    Acct_State : in out ATM_Utilities.Account_State;
package ATM_Session is

-- ———————— EXCEPTIONS

    Unknown_Check           : exception;
    Invalid_Withdraw        : exception;
    Invalid_Transfer        : exception;
    Check_Already_Cleared   : exception;

-- ———————— BASIC   FUNCTIONS

    function Amount_In ( Account : in Account_Type ) return Dollars_Type;

    function Status_Of ( Check : in Check_Number_Type )
        return Check_Status_Type;

    function Amount_Of ( Check : in Check_Number_Type )
        return Positive_Dollars_Type;
--| where Status_Of( Check ) = Not_Yet_Received => raise Unknown_Check;

-- ———————— SUBPROGRAMS

--: function Net_Worth return Dollars_Type;
--| where
--|     return Amount_In( Checking ) + Amount_In( Savings ) - Amount_In( Loan );

    procedure Stop_Payment ( Check : in Check_Number_Type );
--| where Status_Of( Check ) = Cleared => raise Check_Already_Cleared,
--|     out( Status_Of( Check ) = Stopped );

    procedure Deposit ( N_Dollars     : in      Positive_Dollars_Type;
                        Into_Account  : in out  Account_Type);
--| where
--|     out( Amount_In(Into_Account)
--|             = if Into_Account = Loan then
--|                     Amount_In(in Into_Account) - N_Dollars
--|               else
--|                     Amount_In(in Into_Account) + N_Dollars
--|               end if );
```

```
    procedure Withdraw ( N_Dollars      : in      Positive_Dollars_Type;
                         From_Account   : in out  Account_Type);
--| where ( From_Account = Loan and then
--|           Amount_In(Loan) + N_Dollars > Acct_State.Line_Of_Credit ) or else
--|         ( From_Account isin Checking .. Savings and then
--|           N_Dollars > Amount_In(From_Account))
--|              => raise Invalid_Withdraw,
--|     out( Amount_In(From_Account)
--|                = if From_Account = Loan then
--|                            Amount_In(in From_Account) + N_Dollars
--|                  else
--|                            Amount_In(in From_Account) - N_Dollars
--|                  end if );

    procedure Transfer ( N_Dollars      : in      Positive_Dollars_Type;
                         From_Account   : in out  Account_Type;
                         To_Account     : in out  Account_Type);
--| where ( From_Account = Loan  and then
--|           Amount_In(Loan) + N_Dollars > Acct_State.Line_Of_Credit ) or else
--|         ( From_Account isin Checking .. Savings and then
--|           N_Dollars > Amount_In( From_Account ) )
--|              => raise Invalid_Withdraw,
--|         From_Account = To_Account => raise Invalid_Transfer,
--|     out( ATM_Session =
--|                in ATM_Session[ Withdraw( N_Dollars, From_Account ) ;
--|                                Deposit( N_Dollars, To_Account )]);

-- ------- AXIOMS

--| axiom
--|     for all Account : Account_Type;
--|             Check   : Check_Number_Type =>
--|         ATM_Session'Initial.Amount_In(Account) = in Acct_State.Acct_Balances(Account),
--|         ATM_Session'Initial.Status_Of(Check) = in Acct_State.Check_Status(Check),
--|         ATM_Session'Initial.Amount_Of(Check) = in Acct_State.Check_Amount(Check);

end ATM_Session;

-- Commentary:
-- The above packages are two pieces of a larger banking system. We can think of the system as follows.
-- There is a large central database of customer account information, connected to some number of auto-
-- matic teller machines. When a customer uses an ATM, if he has correctly inserted his card and entered
-- his code, a user-interface is instantiated for the ATM session that follows. The customer's account is
-- locked, and the account information is sent to the user-interface. By using the ATM functions, the
-- customer changes the state of his account. When the session ends, the updated state is copied back into
-- the central database and is unlocked.
```

```
-- Accounts are defined in the package ATM_Utilities. Every customer is assumed to have one savings
-- account, one checking account, and a "loan" account with a certain line of credit which he can borrow
-- against (since account definitions are completely encapsulated in this package, this is an easy assumption
-- to change). The state consists of: balances in each of the customer accounts, status of all the checks he
-- has been issued, the values of checks which have been received by the bank, and the current line-of-credit
-- limit on his loan account. A type annotation insures that the loan account balance never exceeds the
-- customer's credit limit.

-- An instance of package ATM_Session exists for the duration of a single customer session. It is instantiated
-- with the current state of the customer's accounts; the axioms define the relationship between the initial
-- value of this generic formal parameter, and the initial values of the observer functions.

-- ATM_Session provides all of the standard ATM user functions. An interesting aspect of this specification
-- is that we are able to define explicitly the difference between a loan account and the other account
-- kinds. Though both loan accounts and savings accounts have positive-valued balances, a savings balance
-- represents the amount of money a customer has, whereas a loan balance represents the amount of money
-- he owes. This means that, for example, a deposit to a loan account is fundamentally different from a
-- deposit to a savings account. The difference is simply and explicitly stated. Also, note that subprogram
-- Transfer is defined in terms of Withdrawal and Deposit; its out-annotation specifies that the state of a
-- customer's accounts after a Transfer of money from one account to another is equivalent to what the
-- state would be if he had withdrawn the money from the source account, and then deposited it in the
-- destination account.
```

## 5.2 Math Functions Package

```
-- Package Name:
--  Math Functions (generic)
-- Description:
--  This package defines commonly used math functions for floating point numbers, such as trigonometric
-- and logarithmic functions.
-- Author:
--  Doug Bryan, Chuan-Chieh Ko


package Floating_Point_Math_Functions is

    Pi          : constant := 3.14159_26535_89793_23846_26433_83279_50288;
    E           : constant := 2.71828_18284_59045_23536_02874_71352_66250;

--: Max_Delta : constant Float := 2.0 / (10.0 ** Float'Digits);

    subtype Amplitude is Float range -1.0 .. 1.0;

    subtype Reciprocal_Amplitude is Float;
--| where
--|     X : Reciprocal_Amplitude   =>   X <= -1.0   or   X >= 1.0;

    subtype Angle_1 is Float range -0.5 * Pi .. 0.5 * Pi;

    subtype Angle_2 is Float range 0.0 .. Pi;

-- ------- EXCEPTIONS

    Math_Functions_Error : exception;

-- ------- PREDEFINED  SUBPROGRAMS

-- function "="    ( Left, Right : Float ) return Boolean;
-- function "/="   ( Left, Right : Float ) return Boolean;
-- function "< "   ( Left, Right : Float ) return Boolean;
-- function "<="   ( Left, Right : Float ) return Boolean;
-- function ">"    ( Left, Right : Float ) return Boolean;
-- function ">="   ( Left, Right : Float ) return Boolean;

-- function "+"    ( Right : Float ) return Float;
-- function "-"    ( Right : Float ) return Float;
-- function "abs"  ( Right : Float ) return Float;

-- function "+"    ( Left, Right : Float ) return Float;
-- function "-"    ( Left, Right : Float ) return Float;
-- function "*"    ( Left, Right : Float ) return Float;
-- function "/"    ( Left, Right : Float ) return Float;

-- function "**"   ( Left : Float; Right : Integer ) return Float;
```

```
-- ------- SUBPROGRAMS

   -- Define equality over floating point numbers, taking into account round-off error.
--: function Equal ( Left, Right : Float ) return Boolean;
--| where
--|    return abs( Left - Right ) / Left < 5.0 * Float( Max_Delta );

--: generic
--:    with function F ( X : Float; I : Integer ) return Float;
--: function Summation_Of_F ( X              : Float;
--:                           Beginning_At   : Integer) return Float;

--| where
--|   return ( if Equal( F( X, Beginning_At ), 0.0 ) then
--|               F( X, Beginning_At )
--|            else
--|               F( X, Beginning_At ) + Summation_Of_F( X, Beginning_At + 1)
--|            end if );

   function Factorial ( N : Natural ) return Natural;
--| where
--|    return ( if N > 0 then N * Factorial( N - 1 )
--|             else 1
--|             end if );

--: function Sin_Summation_Component ( X : Float;
--:                                    I : Integer ) return Float;
--| where
--|    return ((-1.0) ** I * X ** (2 * I + 1)
--|                        / Float( Factorial( 2 * I + 1)));

--: function Sin_Series is
--:    new Summation_Of_F (F => Sin_Summation_Component);

--: function Cos_Summation_Component ( X : Float;
--:                                    I : Integer ) return Float;
--| where
--|    return ((-1.0) ** I * X ** (2 * I) / Float(Factorial(2 * I)));

--: function Cos_Series is
--:    new Summation_Of_F (F => Cos_Summation_Component);

--: function Log_Summation_Component ( X : Float;
--:                                    I : Integer ) return Float;
--| where
--|    return -((-X) ** (I + 1) / Float(I + 1));

--: function Log_Series is
--:    new Summation_Of_F (F => Log_Summation_Component);
```

```
    function Sqrt ( X : Float ) return Float;
--| where
--|       X < 0.0 => raise Math_Functions_Error,
--|       return Y : Float => Equal( X, Y * Y );

    function Sin (X : Float) return Amplitude;
--| where
--|       return
--|           Y : Amplitude =>
--|               exist N : Integer =>
--|                    (-0.5 * Pi <= X + Float(N) * Pi <= 0.5 * Pi) and
--|                    ( Equal( Y, Sin_Series(X + Float(N) * Pi, 0)));

    function Cos (X : Float) return Amplitude;
--| where
--|       return
--|           Y : Amplitude =>
--|               exist N : Integer =>
--|                    (-0.5 * Pi <= X + Float(N) * Pi <= 0.5 * Pi) and
--|                    ( Equal( Y, Cos_Series(X + Float(N) * Pi, 0)));

    function Tan ( X : Float ) return Float;
--| where Cos(X) = 0.0 => raise Math_Functions_Error,
--|       return Y : Float => Equal( Y, Sin( X ) / Cos( X ));

    function Sec ( X : Float ) return Reciprocal_Amplitude;
--| where Cos(X) = 0.0 => raise Math_Functions_Error,
--|       return Y : Reciprocal_Amplitude => Equal( Y, 1.0 / Cos( X ));

    function Csc (X : Float) return Reciprocal_Amplitude;
--| where Sin(X) = 0.0 => raise Math_Functions_Error,
--|       return Y : Reciprocal_Amplitude => Equal( Y, 1.0 / Sin( X ));

    function Cot (X : Float) return Float;
--| where Sin(X) = 0.0 => raise Math_Functions_Error,
--|       return Y : Float => Equal( Y, Cos( X ) / Sin( X ));

    function Arc_Sin (X : Amplitude) return Angle_1;
--| where
--|       return Z : Angle_1 => Equal( X, Sin( Z ));

    function Arc_Cos (X : Amplitude) return Angle_2;
--| where
--|       return Z : Angle_2 => Equal( X, Cos( Z ));

    function Arc_Tan (X : Float) return Angle_1;
--| where
--|       return Z : Angle_1 => Equal( X, Tan( Z ));

    function Arc_Sec (X : Reciprocal_Amplitude) return Angle_2;
--| where
--|       return Z : Angle_2 => Equal( X, Sec( Z ));
```

```
    function Arc_Csc (X : Reciprocal_Amplitude) return Angle_1;
--| where
--|       return Z : Angle_1 => Equal( X, Csc( Z ));

    function Arc_Cot (X : Float) return Angle_2;
--| where
--|       return Z : Angle_2 => Equal( X, Cot( Z ));

    function Log (X : Float) return Float;
--| where X <= 0.0   => raise Math_Functions_Error,
--|       return Y : Float =>
--|           exist N : Integer =>
--|               ( 2.0 / ( E + 1.0 )
--|                    <= X / E**N
--|                    <= 2.0 * E / ( E + 1.0 )) and
--|               ( Equal( Y, Log_Series( X / E ** N - 1.0, 0 )));

    function Exp (X : Float) return Float;
--| where
--|       return Y : Float => Equal( X,  Log( Y ));

    function "**" (X, Y : Float) return Float;
--| where X < 0.0   =>  raise Math_Functions_Error,
--|       return Z : Float =>
--|           Equal( Z, ( if X = 0.0 then 0.0
--|                       else Exp(Y * Log(X))
--|                       end if ) );

--  ———————— A X I O M S

--| axiom
--|       for all A, B, X : Float;
--|                  Y    : Angle_1;
--|                  Z    : Angle_2;
--|                  U    : Amplitude;
--|                  V    : Reciprocal_Amplitude;
--|                  K    : Integer =>
--|
--|              Equal (Tan(X),     Sin(X) / Cos(X)),
--|              Equal (Csc(X),     1.0 / Sin(X)),
--|              Equal (Sec(X),     1.0 / Cos(X)),
--|              Equal (Cot(X),     1.0 / Tan(Z)),

--|              Equal (1.0,        Sin(X)**2 + Cos(X)**2),

--|              Equal (Cos (-X),     Cos (X)),
--|              Equal (Sin (-X),     - Sin (X)),
--|              Equal (Tan (-X),     - Tan (X)),

--|              Equal (Sin(X),      Sin (X + Pi * (2.0 * Float(K)))),
--|              Equal (Cos(X),      Cos (X + Pi * (2.0 * Float(K)))),
--|              Equal (Tan(X),      Tan (X + Pi * (1.0 * Float(K)))),
```

```
--|          Equal (1.0,      Sec(X)**2 - Tan(X)**2),
--|          Equal (1.0,      Csc(X)**2 - Cot(X)**2),

--|          Equal (Sin(A + B), Sin(A)*Cos(B)+Cos(A)*Sin(B)),
--|          Equal (Sin(A - B), Sin(A)*Cos(B)-Cos(A)*Sin(B)),
--|          Equal (Cos(A + B), Cos(A)*Cos(B)-Sin(A)*Sin(B)),
--|          Equal (Cos(A - B), Cos(A)*Cos(B)+Sin(A)*Sin(B)),
--|          Equal (Tan(A + B), (Tan(A)+Tan(B))/(1.0-Tan(A)*Tan(B))),
--|          Equal (Tan(A - B), (Tan(A)-Tan(B))/(1.0+Tan(A)*Tan(B))),

--|          Equal (Sin(2.0 * A),    2.0 * Sin(A) * Cos(A)),
--|          Equal (Cos(2.0 * A),    Cos(A)**2 - Sin(A)**2),

--|          Equal (Sin(0.5 * A)**2,      0.5 * (1.0 - Cos(A))),
--|          Equal (Cos(0.5 * A)**2,      0.5 * (1.0 + Cos(A))),

--|          Equal (Arc_Sin (Sin (Y)),    Y),
--|          Equal (Arc_Cos (Cos (Z)),    Z),
--|          Equal (Arc_Tan (Tan (Y)),    Y),
--|          Equal (Arc_Csc (Csc (Y)),    Y),
--|          Equal (Arc_Sec (Sec (Z)),    Z),
--|          Equal (Arc_Cot (Cot (Z)),    Z),

--|          Equal( Pi / 2.0,       abs (Arc_Tan(X) + Arc_Cot (X))),
--|          Equal( Arc_Csc(V),     Arc_Sin (1.0/V)),
--|          Equal( Arc_Sec(V),     Arc_Cos (1.0/V)),
--|          Equal( Arc_Cot(X),     Arc_Tan (1.0/X)),
--|          Equal( Pi / 2.0,       Arc_Sec(V) + Arc_Csc(V)),
--|          Equal( Arc_Sin (-U), - Arc_Sin (U)),
--|          Equal( Arc_Cos (-U), Pi - Arc_Cos (U)),
--|          Equal( Arc_Tan (-X), - Arc_Tan (X)),
--|          Equal( Arc_Csc (-V), - Arc_Csc (V)),
--|          Equal( Arc_Sec (-V), Pi - Arc_Sec (V)),
--|          Equal( Arc_Cot (-X), Pi - Arc_Cot (X)),

--|          Equal( Log(Exp(X)),   X),
--|          Equal( Exp(Log(X)),   X),
--|
--|          Equal( X ** (A + B),X ** A * X ** B),
--|          Equal( (A * B) ** X,A ** X * B ** X),
--|          Equal( X ** 0.5,      Sqrt(X)),
--|          Equal( X ** (-A),     1.0 / X ** A);

end Floating_Point_Math_Functions;

-- Commentary:

-- An inherent problem of specifying packages which use floating point numbers is accounting for round-
-- off errors in results. In function result annotations, the returnconstruct implies the predefined "="
-- relationship between the returned value and the result expression. So, for example, if the Tan function
-- were annotated as follows:
```

```
    function Tan ( X : Float ) return Float;
--| where Cos(X) = 0.0 => raise Math_Functions_Error,
--|       return Sin( X ) / Cos( X );

-- It is implied that the returned result must be exactly equal to the calculated value of:

   Sin( X ) / Cos( X ).

-- More than likely, this is too restrictive.
-- The Anna specification accounts for this problem by defining Equal, an approximate equality function
-- for floating-point numbers. All result annotations and axioms are defined using that function, except
-- where one argument is the floating-point constant 0.0. It is assumed that there is an exact representation
-- for 0.0, and so we may use the predefined equality when it is involved.
```

## 5.3 Complex Numbers Package

```
-- Package Name:
--  Complex Numbers
-- Description:
--  This package implements complex numbers. It defines both operations and relations over complex num-
-- bers, as well as input/output facilities.
-- Dependency:
--  Math_Functions Package
-- Author:
--  Chuan-Chieh Ko

   with Text_IO,
        Floating_Point_Math_Functions;
   package Complex_Numbers is

      package FPMF renames Floating_Point_Math_Functions;

      type Complex is limited private;

      I : constant Complex;

-- ---------- E X C E P T I O N S
      Order_Not_Defined : exception;
         -- This exception is raised when ordering is not defined for complex numbers.

-- ---------- B A S I C   F U N C T I O N
   -- Return the real part of a complex number.
      function Re ( Z : Complex ) return Float;

   -- Return the imaginary part of a complex number.
      function Im ( Z : Complex ) return Float;

-- ---------- S U B P R O G R A M S
      function Cons ( X, Y : Float ) return Complex;
--|      where
--|        return Z : Complex => FPMF.Equal( X, Re(Z)) and
--|                              FPMF.Equal( Y, Im(Z));

      function "=" ( U, V : Complex ) return Boolean;
--|      where
--|        return FPMF.Equal( Re(U),  Re(V)) and then
--|               FPMF.Equal( Im(U),  Im(V));

      function ">" (U, V : Complex ) return Boolean;
--|      where Im(U) /= 0.0 or Im(V) /= 0.0 => raise Order_Not_Defined,
--|        return Re(U) > Re(V);
```

```
        function ">=" ( U, V : Complex ) return Boolean;
--|        where U /= V and ( Im(U) /= 0.0 or Im(V) /= 0.0 )
--|                                    =>  raise Order_Not_Defined,
--|           return
--|             if ( Im(U) = 0.0 and Im(V) = 0.0 ) then
--|                   Re(U) >= Re(V)
--|             else
--|                   U = V
--|             end if;

        function "<" ( U, V : Complex ) return Boolean;
--|        where Im(U) /= 0.0 or Im(V) /= 0.0 => raise Order_Not_Defined,
--|           return V > U;

        function "<=" ( U, V : Complex) return Boolean;
--|        where U /= V and (Im(U) /= 0.0 or Im(V) /= 0.0)
--|                                    =>  raise Order_Not_Defined,
--|           return V >= U;

        function "abs" ( Z : Complex ) return Float;
--|        where return FPMF.Sqrt( Re(Z) * Re(Z) + Im(Z) * Im(Z));

        function "+" ( U, V : Complex ) return Complex;
--|        where
--|           return Z : Complex =>
--|               FPMF.Equal( Re(Z), Re(U) + Re(V)) and
--|               FPMF.Equal( Im(Z), Im(U) + Im(V));

        function "-" ( U : Complex ) return Complex;
--|        where
--|           return Z : Complex =>
--|               FPMF.Equal( Re(Z), - Re(U)) and
--|               FPMF.Equal( Im(Z), - Im(U));

        function "-" (U, V : Complex) return Complex;
--|        where
--|           return Z : Complex =>
--|               FPMF.Equal( Re(Z), Re(U) - Re(V)) and
--|               FPMF.Equal( Im(Z), Im(U) - Im(V));

        function "*" (U, V : Complex) return Complex;
--|        where
--|           return Z : Complex  =>
--|               FPMF.Equal(Re(Z), Re(U) * Re(V) - Im(U) * Im(V)) and
--|               FPMF.Equal(Im(Z), Im(U) * Re(V) + Re(U) * Im(V));

        function "/" ( U, V : Complex ) return Complex;
--|        where
--|           return Z : Complex  =>
--|               FPMF.Equal( Re(Z), "/"( Re(U) * Re(V) + Im(U) * Im(V),
--|                                       abs V * abs V )) and
--|               FPMF.Equal( Im(Z), "/"( Im(U) * Re(V) - Re(U) * Im(V),
--|                                       abs V * abs V ));
```

```
      function Arg ( Z : Complex ) return Float;
--|      where
--|        return X : Float  =>
--|          if Re(Z) > 0.0 then
--|            if Im(Z) >= 0.0 then
--|                 FPMF.Equal( X, FPMF.Arc_Tan( Im(Z) / Re(Z)))
--|            else
--|                 FPMF.Equal( X, FPMF.Arc_Tan( Im(Z) / Re(Z))
--|                                + 2.0 * FPMF.PI )
--|            end if
--|          elsif Re(Z) = 0.0 then
--|            if Im(Z) > 0.0 then
--|                 FPMF.Equal( X, FPMF.PI / 2.0 )
--|            elsif Im(Z) = 0.0 then
--|                 X = 0.0
--|            else
--|                 FPMF.Equal( X, 3.0 * FPMF.PI / 2.0 )
--|            end if
--|          else
--|            FPMF.Equal( X, FPMF.Arc_Tan( Im(Z) / Re(Z)) + FPMF.PI )
--|          end if;

      function "**" ( Z : Complex;  X : Integer ) return Complex;
--|      where
--|        return W : Complex  =>
--|          if X > 0 then
--|           FPMF.Equal( Re(W), ((abs Z)**X)*FPMF.Cos(Float(X)*Arg(Z))) and
--|           FPMF.Equal( Im(W), ((abs Z)**X)*FPMF.Sin(Float(X)*Arg(Z)))
--|          elsif X = 0 then
--|           W = Cons(1.0, 0.0)
--|          else
--|           FPMF.Equal( Re(W), ((abs Z)**X)/FPMF.Cos(Float(X)*Arg(Z))) and
--|           FPMF.Equal( Im(W), ((abs Z)**X)/FPMF.Sin(Float(X)*Arg(Z)))
--|          end if;

      function "**" ( Z : Complex; X : Float ) return Complex;
--|      where
--|        return W : Complex  =>
--|          if X > 0.0 then
--|            FPMF.Equal( Re(W), FPMF."**"((abs Z),X)
--|                                 *FPMF.Cos(X*Arg(Z))) and
--|            FPMF.Equal( Im(W), FPMF."**"((abs Z),X)
--|                                 *FPMF.Sin(X*Arg(Z)))
--|          elsif X = 0.0 then
--|            W = Cons (1.0, 0.0)
--|          else
--|            FPMF.Equal( Re(W), FPMF."**"((abs Z),X)/FPMF.Cos(X*Arg(Z))) and
--|            FPMF.Equal( Im(W), FPMF."**"((abs Z),X)/FPMF.Sin(X*Arg(Z)))
--|          end if;
```

```
   -- Input procedures for complex numbers
      procedure Get ( File   : in Text_IO.File_Type;
                      Z      : out Complex;
                      Width: in Text_IO.Field := 0);

      procedure Get ( Z      : out Complex;
                      Width  : in Text_IO.Field := 0);

   -- Output procedures for complex numbers
      Default_Fore : Text_IO.Field := 2;
      Default_Aft  : Text_IO.Field := Float'Digits-1;
      Default_Exp  : Text_IO.Field := 3;

      procedure Put ( File   : in Text_IO.File_Type;
                      Z      : in Complex;
                      Fore   : in Text_IO.Field := Default_Fore;
                      Aft    : in Text_IO.Field := Default_Aft;
                      Exp    : in Text_IO.Field := Default_Exp);

      procedure Put ( Z      : in Complex;
                      Fore   : in Text_IO.Field := Default_Fore;
                      Aft    : in Text_IO.Field := Default_Aft;
                      Exp    : in Text_IO.Field := Default_Exp);

   -- ---------- A X I O M S
   --| axiom
   --|     for all U, V : Complex;  X, Y : Float;  M, N : Integer  =>
   --|         I * I * U = -U,
   --|         I = Cons( 0.0, 1.0 ),
   --|         I * I = Cons( -1.0, 0.0 ),

   --|         U >= V  <->  U > V or U = V,
   --|         U <= V  <->  U < V or U = V,

   --|         0.0 <= Arg(U) < 2.0 * FPMF.PI,
   --|         abs U >= abs (Re(U)),
   --|         abs U >= abs (Im(U)),
   --|         U + (-V) = U - V,

   --|         "=" (U ** (M + N),     U ** M * U ** N),
   --|         "=" ((U * V) ** M,     U ** M * V ** M),
   --|         "=" (U ** 2,           U * U),
   --|         "=" (U ** (-M),        Cons (1.0, 0.0) / U ** M),

   --|         "=" (U ** (X + Y),     U ** X * U ** Y),
   --|         "=" ((U * V) ** X,     U ** X * V ** X),
   --|         "=" (U ** 2.0,         U * U),
   --|         "=" (U ** (-X),        Cons (1.0, 0.0) / U ** X),
```

```
--|          "=" (U, Cons (abs U, 0.0)
--|                        * Cons( FPMF.Cos (Arg (U)), FPMF.Sin (Arg (U)))),
--|          "=" (U * V,   Cons( abs U * abs V, 0.0)
--|                             * Cons( FPMF.Cos (Arg(U) + Arg(V)),
--|                                     FPMF.Sin (Arg(U) + Arg(V)) ));

-- ———————— PRIVATE
private
    type Complex is record
                    R, I : Float;
                end record;

    I : constant Complex := (0.0, 1.0);
end Complex_Numbers;

-- Commentary:
-- In re-using the Math_Functions package to define complex numbers, we also want to re-use the approx-
-- imate Equal function defined there. Even though Equal was defined as virtual, it is still visible to other
-- packages inside annotations.
-- Note also that the input-output routines are not annotated at all. If the predefined package Text_IO
-- were annotated with the appropriate concepts, annotations of Get and Put for complex numbers might
-- be possible.
```

## 5.4 Sorting Package

```
-- Package Name:
--  Anna_Sort_Utilities

-- Description:
-- Anna_Sort_Utilities is a generic sorting package. The Sort subprograms will sort a one dimensional
-- array of any component type that supports assignment, equality, and inequality (private types) indexed by
-- discrete type components. The default linear order is ascending order but may be overridden by the user.
-- The default sort algorithm, Quicksort (non-recursive), may also be overridden.

-- Note that the component type can be a record type. The Sort subprograms are not restricted to simple data
-- types. If records are to be sorted, then the formal generic subprogram parameter "<" must be specified with
-- by a linear order, e.g., a function provided as an actual generic subprogram parameter at instantiation.

-- Note also that the component type can be an access type (which can point to other objects, improving sort
-- efficiency). If access types are to be sorted, then the formal generic subprogram parameter "<" must be
-- specified by a linear order. Since access types can be sorted, the Sort routine below can be used to sort
-- limited types and unconstrained types (designated by an access type).

-- For data in which equality does not truly apply (i.e., real types) one can use the Equal function to specify
-- an equality operation. Hence, one can decide that two numbers are "close enough" to be equal.

-- The number of comparisons and exchanges made to sort the array can be returned. These numbers should
-- give some indication on how much work was actually performed by the sorting algorithms. These numbers
-- can also be used to compare the relative efficiency of the sorting algorithms.

-- This package can be used to sort data on external devices. The user should use this package to sort a
-- subset of the external data, then use a merge operation on all sorted subsets. For example, if the system
-- can only hold 1000 components in RAM, but you need to sort 3000 components, bring in components #1-
-- 1000 and sort them using this routine, and then write them to a file. Next do the same with components
-- #1001-2000, and finally with components #2001-3000. Now merge the three sorted files using a merge
-- package.

-- One of the Sort subprograms is a function which can be used to sort an array and test it against another
-- in an inline expression. This can be useful when comparing the contents of two arrays which may be equal,
-- but not at the identical indices. This will be most useful for comparing the equality of sets implemented
-- as arrays.

-- Other Sort subprograms allow the user to maintain the original state of the array by returning a new
-- array that is sorted. These subprograms will be useful in cases where sorting is required, but the original
-- (unsorted) data must be preserved.

-- Author:
--  Geoff Mendal

with SYSTEM; -- predefined package SYSTEM

generic
   type Component_Type is private;   -- type of the data components
   type Index_Type     is (<>);      -- type of array index
```

```
   -- The following generic formal type is required due to Ada's strong typing requirements. The SORT
   -- subprograms cannot handle anonymous array types. This type will match any unconstrained array
   -- type definition (so that array slices can be sorted too).

   type Array_Type is array (Index_Type range <>) of Component_Type;

   -- The following formal subprogram parameter defaults to the predefined "<" operator which will sort
   -- one-dimensional arrays of the type Component_Type in ascending order (by default). If composite
   -- or access types are to be sorted, a selector function must be specified.

   with function "<" ( Left, Right : in Component_Type )
            return Boolean is <>;

   -- The following formal subprogram parameter defaults to the predefined "=" operator. If user-defined
   -- equality is desired, one can write an equality function and specify it here.

   with function Equal (Left, Right: in Component_Type)
            return Boolean is "=";

   -- The annotations below formally specify assumptions about the above generic formals that must be
   -- satisfied in order to perform correct sorting. These are the axioms of partial order, equality, and
   -- associated relations.

   --| for all X, Y, Z : Component_Type =>
   --|    (not (X < X)) and
   --|    ((X < Y) and (Y < Z) -> (X < Z)) and

   --|    Equal (X, X) and
   --|    (Equal (X, Y) -> Equal (Y, X)) and
   --|    (Equal (X, Y) and Equal (Y, Z) -> Equal (X, Z)) and

   --|    (Equal (X, Y) and (X < Z) -> (Y < Z)) and
   --|    (Equal (X, Y) and (Z < X) -> (Z < Y)) and
   --|    (Equal (X, Y) xor (X < Y) xor (Y < X));

package Anna_Sort_Utilities is

   -- Users can specify the type of sorting algorithm they want by specifying an enumeration literal from
   -- the type below. The default algorithm, Quicksort (non-recursive), generally performs best.

   -- One note about stability of the algorithms: only the Bubble Sorts and Insertion Sort are stable
   -- algorithms. Thus, they are the only algorithms that preserve the ordering of equal components without
   -- use of a selector function. In all cases, a selector function may be specified to introduce stability into
   -- the sorting algorithms.

   type Sort_Algorithm_Type is (Quicksort, Recursive_Quicksort, Bsort,
     Bubble_Sort, Bubble_Sort_with_Quick_Exit, Selection_Sort, Heapsort,
     Insertion_Sort, Merge_Sort);

   -- Quicksort: O(NlogN). Is most efficient when used with large, unsorted arrays. Uses an explicit stack
   -- to maintain state and partitions. Instable. This is the default algorithm.
```

```
-- Recursive_Quicksort: O(NlogN). Is most efficient when used with large, unsorted arrays. Recursive
-- nature may introduce significant memory overhead for very large arrays. Instable.

-- Bsort: O(NlogN). Is most efficient when used with large arrays that are already sorted, partially
-- sorted, or sorted in reverse. Recursive. Instable.

-- Bubble_Sort: O(N**2). Is most efficient when used with small arrays that are almost already sorted.
-- Non-recursive. Brute force. Low memory requirements. Stable.

-- Bubble_Sort_with_Quick_Exit: O(N**2). Is most efficient when used with small arrays that are
-- almost already sorted. Non-recursive. Same as bubble sort above except brute force is limited. Stable.

-- Selection_Sort: O(N**2). Is most efficient when used with small arrays in which the Component_Type
-- is a record type. Non-recursive. Brute force. Instable.

-- Heapsort: O(NlogN). Is most efficient when used with large, unsorted arrays. Non-recursive. Very
-- low memory requirements. Instable.

-- Insertion_Sort: O(N**2). Is most efficient when used with small arrays that are almost already
-- sorted. Non- recursive. Brute force. Stable.

-- Merge_Sort: O(NlogN). Is most efficient when used with medium-large arrays. Non-recursive. In-
-- stable. Uses an auxiliary array to perform merging.

-- The following type declaration should be used to specify the instrumentation analysis results that can
-- be returned by the Sort subprograms below. -1 is only returned if an overflow in calculations has
-- occurred. The Sort subprograms will still sort the array if an overflow in instrumentation analysis
-- data calculations occurs.

type Performance_Instrumentation_Type is range -1 .. SYSTEM.MAX_INT;

--  The following exception is raised during execution of the Sort subprograms which take two arrays as
-- parameters, if these two arrays are not of the same length.

Sort_Arrays_Length_Mismatch : exception;

-- The following virtual functions define the semantics of sorting.
--: function "+" ( L : in Index_Type; R : in Integer )
--:     return Index_Type;

--: function "-" ( L : in Index_Type; R : in Integer )
--:     return Index_Type;

--: function Ordered ( A : in Array_Type ) return Boolean;
--|   where return
--|             (( A'Length <= 1 ) or else
--|              ((( A( A'First ) < A( A'First + 1 )) or
--|                 ( Equal( A( A'First ), A( A'First + 1 )))) and
--|               Ordered( A( A'First+1 .. A'Last ))));
```

```
--: function Permutation ( A, B : in Array_Type ) return Boolean;
--|    where A'Length = B'Length,
--|          return
--|             ( A'Length = 0 ) or else
--|             ( exist I : B'Range =>
--|                   Equal( A( A'First ), B( I )) and
--|                   Permutation( A( A'First + 1 .. A'Last ),
--|                                B( B'First .. I - 1 ) &
--|                                B( I + 1 .. B'Last )));

--: function Stable_Algorithm (A : in Array_Type) return Boolean;

-- The following procedure will sort a one dimensional array of components. It can sort in ascend-
-- ing/descending order or any user-defined order. It can sort components of any type that support
-- equality, inequality, and assignment (private types). The array indices can be of any discrete type.
-- The number of comparisons and exchanges can also be returned.

procedure Sort (
   Sort_Array                : in out Array_Type;
   Number_of_Comparisons,
   Number_of_Exchanges       :    out Performance_Instrumentation_Type;
   Sort_Algorithm            : in     Sort_Algorithm_Type := Quicksort);
   --| where
   --|       raise Sort_Arrays_Length_Mismatch => False,
   --|       out Number_of_Comparisons'Defined,
   --|       out Number_of_Exchanges'Defined,
   --|       out Ordered (Sort_Array),
   --|       out Permutation (in Sort_Array, Sort_Array),
   --|       out Stable_Algorithm (in Sort_Array);

-- The following overloading of procedure Sort should be specified when no instrumentation analysis
-- data are required.

procedure Sort (
   Sort_Array      : in out Array_Type;
   Sort_Algorithm  : in     Sort_Algorithm_Type := Quicksort);
   --| where
   --|       raise Sort_Arrays_Length_Mismatch => False,
   --|       out Ordered (Sort_Array),
   --|       out Permutation (in Sort_Array, Sort_Array),
   --|       out Stable_Algorithm (in Sort_Array);
```

```
   -- The following overloading of procedure Sort should be used when the original data must be preserved
   -- and instrumentation analysis results are required.

    procedure Sort (
      Unsorted_Array            : in       Array_Type;
      Sorted_Array              :    out   Array_Type;
      Number_of_Comparisons,
      Number_of_Exchanges       :    out   Performance_Instrumentation_Type;
      Sort_Algorithm            : in       Sort_Algorithm_Type := Quicksort);
      --| where
      --|         Unsorted_Array'Length /= Sorted_Array'Length =>
      --|           raise Sort_Arrays_Length_Mismatch,
      --|         out Number_of_Comparisons'Defined,
      --|         out Number_of_Exchanges'Defined,
      --|         out Ordered (Sorted_Array),
      --|         out Permutation (Unsorted_Array, Sorted_Array),
      --|         out Stable_Algorithm (Unsorted_Array);

   -- The following overloading of procedure Sort should be used when the original data must be preserved
   -- and no instrumentation analysis results are required.

   procedure Sort (
     Unsorted_Array : in       Array_Type;
     Sorted_Array   :    out   Array_Type;
     Sort_Algorithm : in       Sort_Algorithm_Type := Quicksort);
     --| where
     --|         Unsorted_Array'Length /= Sorted_Array'Length =>
     --|           raise Sort_Arrays_Length_Mismatch,
     --|         out Ordered (Sorted_Array),
     --|         out Permutation (Unsorted_Array, Sorted_Array),
     --|         out Stable_Algorithm (Unsorted_Array);

   -- The following overloading of function Sort should be used when sorting is required in an inline
   -- expression.

   function Sort (
     Sort_Array      : in Array_Type;
     Sort_Algorithm  : in Sort_Algorithm_Type := Quicksort)
     return Array_Type;
     --| where
     --|         raise Sort_Arrays_Length_Mismatch => False,
     --|         return A : Array_Type =>
     --|           Ordered (A) and
     --|           Permutation (Sort_Array, A) and
     --|           Stable_Algorithm (Sort_Array);

end Anna_Sort_Utilities;
```

# Chapter 6

# System Examples

This chapter presents four complex examples of Anna package specification. These are included to show the use of Anna on large, real world packages, and to suggest possible extensions of Anna for handling complex examples. The examples also illustrate more complex specification methodologies. Examples include:

- Library Package – A model of a book library. The library package uses the set package to simplify the specification.
- Petri Nets Package
- Mutual Exclusion – A model of a two task mutual exclusion problem; it uses the Petri Nets package.
- Ada Logic Interface – A collection of packages that define an Ada interface to a general purpose logic package.

## 6.1 Library Book Package

```
-- Package Name:
--  Library
-- Description:
--  This package implements a library management system.
-- Dependency:
--  Set_Package
-- Author:
--  David Luckham
-- Source:
--  Problem Set for the 4th International Workshop on Software ACM SIGSOFT Software
--  Engineering Notes, Vol 11,


   with Set_Package;

   package Library is

      type Book_Rec is private;
      type Book is access Book_Rec;

      package Set_Of_Books is new Set_Package( Book );
      type Book_Set is new Set_Of_Books.Set;
--:   use Set_Of_Books;

      type Password is limited private;

      type User_Kinds is ( Staff, Borrower );
      type Status is ( Available, Checked_Out, Not_Available );

      subtype User      is String(1 .. 255);
      subtype Author    is String(1 .. 255);
      subtype Title     is String(1 .. 255);
      subtype Subject   is String(1 .. 255);

      Book_Limit : constant Positive := Positive'Last;

-- ---------- EXCEPTIONS

      Unavailable                 : exception;
      Book_Available              : exception;
      Staff_Password_Required     : exception;
      Incorrect_Password          : exception;
      Limit_Exceeded              : exception;

-- ---------- VIRTUAL   CONCEPTS

--:   function Library_Books return Book_Set;

      package Set_Of_Authors is new Set_Package(Author);
      type Author_Set is new Set_Of_Authors.Set;
--:   use Set_Of_Authors;
```

```
-- Attributes of Books

--:  function Last_Borrower ( B : Book ) return User;

--:  function Status_Of ( B : Book ) return Status;
--|     where
--|       return S : Status =>
--|         not Is_In( B, Library_Books ) -> S = Not_Available;

--:  function Borrower ( B : Book ) return User;
--|     where Is_In( B, Library_Books ),
--|           Status_Of(B) = Available => raise Book_Available,
--|       return Last_Borrower( B );

    -- Attribute of User

--:  function Borrowed ( U : User ) return Book_Set;
--|     where
--|       return S : Book_Set =>
--|         for all B : Book =>
--|           Is_In(B, S) <-> ( Status_Of(B) = Checked_Out and then
--|                             Borrower(B) = U );

    -- Attributes of Passwords (User_Kind, User)
--:  function User_Kind ( P : Password ) return User_Kinds;

--:  function User_Of ( P : Password ) return User;

-- ———————— SUBPROGRAMS

    function Author_Of ( B : Book ) return Author_Set;
--|    where Is_In( B, Library_Books );

    function Title_Of ( B : Book ) return Title;
--|    where Is_In( B, Library_Books );

    function Subject_Of ( B : Book ) return Subject;
--|    where Is_In( B, Library_Books );

    procedure Check_Out ( B : Book; U : User; P : Password );
--|    where
--|      User_Kind(P) /= Staff => raise Staff_Password_Required,
--|      Status_Of(B) /= Available => raise Unavailable,
--|      Cardinality(Borrowed(U)) >= Book_Limit => raise Limit_Exceeded,
--|      out( Status_Of( B ) = Checked_Out ),
--|      out( Last_Borrower( B ) = U ),
--|      out( Borrowed( U ) = in Borrowed( U ) + B );

    procedure Return_Book ( B : Book; U : User; P : Password );
--|    where Status_Of(B) = Checked_Out,
--|      User_Kind(P) /= Staff => raise Staff_Password_Required,
--|      out( Status_Of(B) = Available ),
--|      out( Borrowed( U ) = in Borrowed( U ) - B );
```

```
      procedure Add ( B : Book; P : Password );
--|      where
--|        User_Kind(P) /= Staff => raise Staff_Password_Required,
--|        out( Status_Of(B) = Available ),
--|        out( Library_Books = in Library_Books + B );

      procedure Remove ( B : Book; P : Password );
--|      where
--|        User_Kind(P) /= Staff => raise Staff_Password_Required,
--|        out( Status_Of(B) = Not_Available ),
--|        out( Library_Books =  in Library_Books - B );

      function Retrieve_Subject ( S : Subject ) return Book_Set;
--|      where
--|        return BS : Book_Set =>
--|          for all B : Book =>
--|            Is_In( B, BS ) <-> Subject_Of( B ) = S;

      function Retrieve_Author ( A : Author ) return Book_Set;
--|      where
--|        return BS : Book_Set =>
--|          for all B : Book =>
--|            Is_In( B, BS ) <-> Is_In( A, Author_Of( B ));

      function Retrieve ( S : User; P : Password ) return Book_Set;
--|      where User_Kind(P) /= Staff  or User_Of(P) /= S =>
--|                raise Incorrect_Password,
--|        return BS : Book_Set => BS = Borrowed( S );

      function Retrieve ( B : Book; P : Password ) return User;
--|      where User_Kind(P) /= Staff => raise Staff_Password_Required,
--|        return Last_Borrower( B );

-- ———————— AXIOMS

--|   axiom
--|     for all St : Library'Type; B : Book; U : User =>
--|       Is_In(B, St.Library_Books) ->
--|         ( St.Status_Of(B) = Available or
--|           St.Status_Of(B) = Checked_Out   ),
--|       Cardinality(St.Borrowed(U)) <= Book_Limit;

      private

        type Book_Impl;
        type Book_Rec is access Book_Impl;

        type Password_Rec;
        type Password is access Password_Rec;

      end Library;
```

## 6.2 Petri Net Interpreter Package

```
-- Package Name:
--  Petri Net Interpreter (generic)

-- Description:
--  Generic package for defining and manipulating Petri-Nets.
--  PETRI_NET_INTERPRETER provides an abstract set of Petri Net manipulation routines. The state
-- of the package is simply the current state of the input Petri Net. The terminology is from Peterson's
-- article in "Computing Surveys", Vol. 9, No. 3, September, 1977.

-- Basic terminology:
--  Places ↔ Nodes
--  Transitions ↔ Events
--  Markings ↔ Tokens

-- A Petri Net is a theoretical model of distributed processing and control systems. A Petri Net is comprised
-- of a collection of places and transitions; transitions may be defined by listing their input places and output
-- places. Each place may have an arbitrary number of markings. A graphical representation of Petri Nets
-- uses circles for places, lines for transitions, input constraints represented by edges directed from places
-- to transitions and output constraints represented by edges directed from transitions to places. Thus, a
-- directed path in a graphical representation of a Petri Net visits an alternating sequence of places and
-- transition. The state or configuration of a Petri Net may be described by specifying the set of places,
-- transitions and current set of markings. Only the placement of markings may change as the Petri Net is
-- manipulated.

-- Petri Nets are executed on a selected transition; execution is defined formally below and is considered to
-- be an atomic operation. Also of interest is the reachability of one configuration from another.

-- Author:
--  David S. Rosenblum

generic

-- The generic formal part of this package represents a single Petri Net which is executed and maintained.
-- A discrete Place_Type and Transition_Type are specified, and Transitions are specified as sets of input
-- and output Places obtainable from functions. Finally, the initial configuration of the input net is supplied
-- by a function which specifies the initial number of tokens at each place.

   type Place_Type   is (<>);
   type Transition_Type is (<>);

   -- Transitions are defined in terms of sets of places:

   type Set is limited private;
   with function Is_Member (P : in Place_Type; S : in Set) return Boolean;
   with function Is_Empty (S : in Set) return Boolean;

--| << Minimum_Expectations >>
--| for all S : Set =>
--|     Is_Empty(S) <-> (for all P : Place_Type => not Is_Member(P, S));
```

```
      -- Following are functions defining the structure of the net. Since Set is a limited type, the following two
      -- functions may be called only when the call appears as an actual parameter of some other subprogram
      -- with an in mode formal parameter of type Set.

      with function Inputs_Of  (T : in Transition_Type) return Set;

--| << Inputs_Substitutivity >>
--| for all T1, T2 : Transition_Type =>
--|       (T1 = T2)  ->  (Inputs_Of (T1) = Inputs_Of (T2));

      with function Outputs_Of (T : in Transition_Type) return Set;

--| << Outputs_Substitutivity >>
--| for all T1, T2 : Transition_Type =>
--|       (T1 = T2)  ->  (Outputs_Of (T1) = Outputs_Of (T2));

-- Function defining the initial configuration of the input net:

      with function Initial_Tokens_At (P : in Place_Type) return Natural;

--| << Weak_Connectivity_Constraints >>
--| for all T : Transition_Type =>
--|       not (Is_Empty (Inputs_Of (T)) or Is_Empty (Outputs_Of (T))),
--| for all P : Place_Type =>
--|       (exist T : Transition_Type =>
--|           Is_Member (P, Inputs_Of (T)) or Is_Member (P, Outputs_Of (T)));

package Petri_Net_Interpreter is

-- --------EXCEPTIONS

      Unreachable : exception;

-- --------SUBPROGRAMS

      function Tokens_At (P : in Place_Type) return Natural;
--| where for all P : Place_Type =>
--|                 Petri_Net_Interpreter'Initial.Tokens_At (P)
--|                     = Initial_Tokens_At (P),
--|       out (Petri_Net_Interpreter'State = in Petri_Net_Interpreter'State);
          -- A basic function returning current number of markings at place P.

      function Enabled (T : in Transition_Type) return Boolean;
--| where
--|       return for all P : Place_Type =>
--|            Is_Member (P, Inputs_Of (T)) -> Tokens_At (P) > 0,
--|       out (Petri_Net_Interpreter'State = in Petri_Net_Interpreter'State);
```

```
      procedure Execute (T : in Transition_Type);
--|  where not Enabled (T) => raise Unreachable,
--|        raise Unreachable => Petri_Net_Interpreter'State
--|                                  = in Petri_Net_Interpreter'State,
--|      out (for all P : Place_Type =>
--|                Tokens_At (P) =
--|                     if ( Is_Member (P, Inputs_Of  (T)) and
--|                          Is_Member (P, Outputs_Of (T))) then
--|                       in Petri_Net_Interpreter.Tokens_At (P)
--|                     elsif ( Is_Member (P, Inputs_Of (T))) then
--|                       in Petri_Net_Interpreter.Tokens_At (P) - 1
--|                     elsif ( Is_Member (P, Outputs_Of (T))) then
--|                       in Petri_Net_Interpreter.Tokens_At (P) + 1
--|                     else
--|                       in Petri_Net_Interpreter.Tokens_At (P)
--|                     end if );

-- Actual state type introduced for reachability queries:

      type Petri_Net_State is limited private;

      function "=" (Left, Right : Petri_Net_State) return Boolean;

-- Primitive state constructors:

      function Initial_State return Petri_Net_State;
--|  where
--|      out (Petri_Net_Interpreter'State = in Petri_Net_Interpreter'State);

      function Current_State return Petri_Net_State;
--|  where
--|      out (Petri_Net_Interpreter'State = in Petri_Net_Interpreter'State);

      generic
         with function State_Tokens_At (P : Place_Type) return Natural;
         --| << Token_Substitutivity >>
         --| for all P1, P2 : Place_Type =>
         --|       (P1 = P2)  ->  (State_Tokens_At (P1) = State_Tokens_At (P2));
      function Equivalent_State return Petri_Net_State;
--|  where
--|      out (Petri_Net_Interpreter'State = in Petri_Net_Interpreter'State),
--|      return P : Petri_Net_State =>
--|          for all S : Petri_Net_Interpreter'Type =>
--|              S.Current_State = P ->
--|                  (for all Pl : Place_Type => S.Tokens_At (Pl) = State_Tokens_At (Pl));

      function Reachable (Target, From : in Petri_Net_State) return Boolean;
--|  where
--|      return exist T : Transition_Type; Sp : Petri_Net_Interpreter'Type =>
--|                    Sp.Current_State = From and then
--|                    ( Sp [Execute (T)].Current_State = Target or else
--|                      Reachable (Target, Sp [Execute (T)].Current_State ));
```

```
-- ———————— A X I O M S

--|    axiom
--|      for all T : Transition_Type;
--|              P : Place_Type;
--|              S1, S2, S3 : Petri_Net_State;
--|              PN1, PN2 : Petri_Net_Interpreter'Type =>
--|        Reachable (S2, S1) and Reachable (S3, S2) -> Reachable (S3, S1),
--|        PN1.Initial_State   = PN2.Initial_State,
--|        PN1.Current_State = PN2.Current_State <-> PN1 = PN2,
--|        PN1.Current_State = PN1.Initial_State <-> PN1 = Petri_Net_Interpreter'Initial,
--|        Initial_Tokens_At (P) = 0 and
--|          not Is_Member (P, Outputs_Of (T)) ->
--|            (not exist   S4 : Petri_Net_State;
--|                         PN5 : Petri_Net_Interpreter'Type =>
--|                 PN5.Current_State = S4 and
--|                 PN5.Tokens_At (P) > 0 and
--|                 Reachable (S4, Initial_State));

    private
      type Petri_Net_State is array (Place_Type) of Natural;
    end Petri_Net_Interpreter;

-- Commentary:
-- This package defines an abstract type representing the state of the petri net. It is defined so that an
-- actual user of the package can compare two states of a petri net. The predefined Anna package state
-- type is not adequate for use in this case because it is not visible to actual code which uses the package.
-- Note that the package assumes the petri net type has already been completely defined and a petri net
-- has been built. These definitions would reside in another package, such as the mutual exclusion example
-- which follows.
```

## 6.3 Mutual Exclusion Model Package

```
-- Package Name:
--  Mutual Exclusion Model
-- Description:
--  Mutual Exclusion Model is a package which builds a petri net model of the mutual exclusion problem
-- with two processes. Axioms are given which specify a valid model of the mutual exclusion problem.
-- Dependency:
--  Packages Set_Package, Petri_Net_Interpreter.
-- Author:
--  David S. Rosenblum

with Set_Package,
     Petri_Net_Interpreter;

package Mutual_Exclusion_Model is

     type Mutex_Places is ( Process_1_Safe_Region,
                            Process_1_Critical_Region,
                            Process_2_Safe_Region,
                            Process_2_Critical_Region,
                            Semaphore );

     type Mutex_Transitions is ( Process_1_Safe_To_Critical,
                                 Process_1_Critical_To_Safe,
                                 Process_2_Safe_To_Critical,
                                 Process_2_Critical_To_Safe );

     package Mutex_Place_Sets is new Set_Package( Mutex_Places );
     use Mutex_Place_Sets;

-- ---------- SUBPROGRAMS

     function Initial_Tokens_At ( P : Mutex_Places ) return Natural;
--|   where
--|       return
--|           if P = Process_1_Safe_Region or P = Process_2_Safe_Region or P = Semaphore
--|           then 1
--|           else 0
--|           end if;
```

```
function Inputs_Of ( Mt : in Mutex_Transitions ) return Mutex_Place_Sets.Set;
--| where
--|     return S : Mutex_Place_Sets.Set =>
--|         if Mt = Process_1_Safe_To_Critical then
--|                 Is_In (Process_1_Safe_Region, S) and
--|                 not Is_In (Process_1_Critical_Region, S) and
--|                 not Is_In (Process_2_Safe_Region, S) and
--|                 not Is_In (Process_2_Critical_Region, S) and
--|                 Is_In (Semaphore, S)
--|         elsif Mt = Process_1_Critical_To_Safe then
--|                 not Is_In (Process_1_Safe_Region, S) and
--|                 Is_In (Process_1_Critical_Region, S) and
--|                 not Is_In (Process_2_Safe_Region, S) and
--|                 not Is_In (Process_2_Critical_Region, S) and
--|                 not Is_In (Semaphore, S)
--|         elsif Mt = Process_2_Safe_To_Critical then
--|                 not Is_In (Process_1_Safe_Region, S) and
--|                 not Is_In (Process_1_Critical_Region, S) and
--|                 Is_In (Process_2_Safe_Region, S) and
--|                 not Is_In (Process_2_Critical_Region, S) and
--|                 Is_In (Semaphore, S)
--|         else -- Mt = Process_2_Critical_To_Safe
--|                 not Is_In (Process_1_Safe_Region, S) and
--|                 not Is_In (Process_1_Critical_Region, S) and
--|                 not Is_In (Process_2_Safe_Region, S) and
--|                 Is_In (Process_2_Critical_Region, S) and
--|                 not Is_In (Semaphore, S)
--|         end if;
```

```
      function Outputs_Of (Mt : in Mutex_Transitions) return Mutex_Place_Sets.Set;
--|    where
--|       return S : Mutex_Place_Sets.Set =>
--|           if Mt = Process_1_Safe_To_Critical then
--|                 not Is_In (Process_1_Safe_Region, S) and
--|                 Is_In (Process_1_Critical_Region, S) and
--|                 not Is_In (Process_2_Safe_Region, S) and
--|                 not Is_In (Process_2_Critical_Region, S) and
--|                 not Is_In (Semaphore, S)
--|           elsif Mt = Process_1_Critical_To_Safe then
--|                 Is_In (Process_1_Safe_Region, S) and
--|                 not Is_In (Process_1_Critical_Region, S) and
--|                 not Is_In (Process_2_Safe_Region, S) and
--|                 not Is_In (Process_2_Critical_Region, S) and
--|                 Is_In (Semaphore, S)
--|           elsif Mt = Process_2_Safe_To_Critical then
--|                 not Is_In (Process_1_Safe_Region, S) and
--|                 not Is_In (Process_1_Critical_Region, S) and
--|                 not Is_In (Process_2_Safe_Region, S) and
--|                 Is_In (Process_2_Critical_Region, S) and
--|                 not Is_In (Semaphore, S)
--|           else -- Mt = Process_2_Critical_To_Safe
--|                 not Is_In (Process_1_Safe_Region, S) and
--|                 not Is_In (Process_1_Critical_Region, S) and
--|                 Is_In (Process_2_Safe_Region, S) and
--|                 not Is_In (Process_2_Critical_Region, S) and
--|                 Is_In (Semaphore, S)
--|           end if;

      package Mutex_Net_Interpreter is
           new Petri_Net_Interpreter ( Mutex_Places,
                                       Mutex_Transitions,
                                       Mutex_Place_Sets.Set,
                                       Mutex_Place_Sets.Is_In,
                                       Mutex_Place_Sets.Is_Empty,
                                       Inputs_Of,
                                       Outputs_Of,
                                       Initial_Tokens_At );
      use Mutex_Net_Interpreter;

      function Unprotected_State_1_Tokens (P : Mutex_Places) return Natural;
--|    where
--|       return
--|           if P = Process_1_Critical_Region or P = Process_2_Critical_Region
--|           then 1
--|           else 0
--|           end if;

      function Unprotected_State_1 is
           new Equivalent_State (Unprotected_State_1_Tokens);
```

```
      function Unprotected_State_2_Tokens (P : Mutex_Places) return Natural;
--|    where
--|        return
--|            if P = Process_1_Safe_Region or P = Process_2_Critical_Region or
--|               P = Semaphore
--|            then 1
--|            else 0
--|            end if;

      function Unprotected_State_2 is
         new Equivalent_State (Unprotected_State_2_Tokens);

      function Unprotected_State_3_Tokens (P : Mutex_Places) return Natural;
--|    where
--|        return
--|            if P = Process_1_Critical_Region or P = Process_2_Safe_Region or
--|               P = Semaphore
--|            then 1
--|            else 0
--|            end if;

      function Unprotected_State_3 is
         new Equivalent_State (Unprotected_State_3_Tokens);

--: function Initial_State is
--:    new Equivalent_State (Initial_Tokens_At);

-- ———————— A X I O M S

--| axiom
--|      for all M : Mutual_Exclusion_Problem'Type;
--|              S : Mutex_Net_Interpreter'Type;
--|              P : Mutex_Places =>
--|          0 <= Mutex_Net_Interpreter.Tokens_At (P) <= 1,
--|          Initial_State = Mutex_Net_Interpreter.Initial_State,
--|          not Reachable (Unprotected_State_1, Initial_State),
--|          not Reachable (Unprotected_State_2, Initial_State),
--|          not Reachable (Unprotected_State_3, Initial_State);

end Mutual_Exclusion_Model;
```

## 6.4 Ada Logic Package

The following example [8] is an example of a set of packages which define a software concept too complex to be effectively represented by a single package. Starting with some basic data types, the author builds a hierarchy of concepts, each using one or more of the previously defined concepts, such that a complex theory (in this case the semantics of logic programming) can be specified with surprising ease.

Basing the description on a hierarchy lends itself to a specific method of annotation. First, individual subprograms are given propagation annotations, which define restrictions on the actual parameters passed to a subprogram. Then Anna axioms, usually a series of universally quantified equations, describe the semantics of subprograms in the package.

Two kinds of packages are used in this set. The first defines new abstract data types and operations on them. The second defines no new types, but does define concepts in terms of existing abstract data types.

Data type packages define one or more abstract data types; usually the types are private, since operations on them are independent of a specific implementation. The operations consist of constructors (or subprograms which build objects of the type) and selectors (subprograms which allow the user to examine objects of the type). For these packages, axioms specify the semantics of selector functions in terms of the constructor functions. For example, in the List_Package, Length and Get_Item are defined in terms of Create and Append. Identifier_Package, List_Package, and Clause_Package are examples of abstract data type packages.

Concept packages define subprograms which use previously defined data types to describe complex concepts. For example, Query_Package uses data types like Clauses and Lists_Of_Clauses, and the concept of Unification, to describe SLD, or Prolog-like resolution. Functions that support these advanced concepts are defined in this set of packages, by highly recursive axioms describing the concept in terms of simpler concept packages. Unification_Package and Query_Package are examples of this kind of package.

Substitution_Package combines the two kinds, both defining a new structure, the Substitution, and a new concept, Applying the Substitution to a Clause.

### 6.4.1 Identifier Package

```
-- Package Name:
--  Identifier_Package
-- Description:
--  This package defines various types of identifiers.
-- Dependency:
--  (none)
-- Author:
--  Neel Madhav

   package IDENTIFIER_PACKAGE is

      type Identifier is private;

      -- Identifiers are the basic building blocks of a clause. They can be constants, variables or integers.
      -- These are mutually exclusive classes. Constants act as predicates of arbitrary arity, but the
      -- pre-defined predicates have fixed aritys.

      function Integer_Identifier ( id : Identifier ) return Boolean;

      function Constant_Identifier ( id : Identifier ) return Boolean;
--|   where
--|      return ( not Integer_Identifier( id ));
```

```
        function Variable_Identifier ( id : Identifier ) return Boolean;
--|     where
--|         return ( not Constant_Identifier( id ) and not Integer_Identifier( id ));

        function Built_in_Predicate ( id : Identifier ) return Boolean;
--|     where Constant_Identifier( id );

        function Built_in_Arity ( id : Identifier ) return Natural;
--|     where Built_in_Predicate( id );

-- The above subprogram annotations constrain the various classes of identifiers to be mutually exclusive
-- and the built-in-predicates to be constants of fixed arity.

-- The above functions and Equality are a complete set of observers for identifiers.

        function Build_Identifier ( s : String ) return Identifier;
        -- Build_Identifier is the only generator for identifiers.

private
     type Identifier_Rec;
     type Identifier is access Identifier_Rec;
end IDENTIFIER_PACKAGE;

-- Commentary:
-- For specific implementations, axioms would also be given to list all the built-in predicates and their
-- arities, e.g.:
--|axiom
--|     Built_In_Predicate( Build_Identifier( "equal" )),
--|     Built_In_Arity( Build_Identifier( "equal" )) = 2;
-- Perhaps the author has overcommitted himself by providing only one generator; this implies that integers,
-- variables, and constants are distinguishable by their string representations, which restricts the generality
-- of the specification. Further, exactly how string structure determines what subclass of Identifier would
-- be built is not annotated (in one implementation of this specification which supports Prolog, Variables
-- and Constants are distinguishable because a Constant identifier begins with a capital letter). A better
-- design might be to have three generator functions, Build_Integer, Build_Variable, and Build_Constant,
-- to abstract out the properties that determine an Identifier's subclass.
```

### 6.4.2 Clause Package

```
-- Package Name:
--  Clause_Package
-- Description:
--  This package defines and provides operations on clauses. There are two types of clauses: facts and rules.
-- Facts are literals like: p( t1, ... , tn ), where p is a predicate and t1 . . tn are terms. Rules are of the
-- form : a1,... ,an :- b1,... bm. The symbol :- has a standard interpretation: is implied by. Commas
-- stand for and's above, and the a's and b's are facts.
-- Dependency:
--  List and Identifier packages.
-- Author:
--  Neel Madhav
```

```
with LIST, IDENTIFIER_PACKAGE; use IDENTIFIER_PACKAGE;

package CLAUSE_PACKAGE is

    type Clause_Rec is limited private;
    type Clause is access Clause_Rec;

    function Equal ( C1, C2 : Clause ) return Boolean;
        -- Is C1 structurally equal to C2? Variables with different names are treated differently.

    procedure Copy ( from : in Clause; into : in out Clause );
        -- Replicate the clause C1 and return it.

    NO_SONS : exception;
        -- This exception is raised by Build_Fact if the predicate is not of the appropriate type.

    ATTR_ERROR : exception;
        -- Raised in various situations where an inappropriate attribute of a node is assigned or queried.

    function Is_Rule ( C : Clause ) return Boolean;
        -- Is the clause a rule? If not, it must be a fact.

    package SON_LIST_PACKAGE is new LIST ( Item => Clause );
    use SON_LIST_PACKAGE;

    type List_Of_Facts is new SON_LIST_PACKAGE.List;
--| where L : List_Of_Facts =>
--|       (for all N : 1 .. Length_Of(L) => not Is_Rule( Get_Item(L,N)));

    No_Facts : constant List_Of_Facts
                    := List_Of_Facts( SON_LIST_PACKAGE.Null_List );

        -- The List_Of_Facts data-structure is used to hold a list of facts. It stands for a conjunction of
        -- those facts when it is the head or tail of a rule. It stands for a list of sons, in order, when it
        -- acts as a son-list of a fact.

    function Get_Identifier ( C : Clause ) return Identifier;
--| where Is_Rule(C) => raise ATTR_ERROR;
        -- For a fact of the form p( t1, ... , tn ) this returns p.

    function Get_Son_List( C : Clause ) return List_Of_Facts;
--| where Is_Rule(C) => raise ATTR_ERROR;
        -- For a fact of the form p( t1, ... , tn ) this returns the list t1, ... , tn.

-- Get_Identifier and Get_Son_List are a complete set of observer functions on facts.
```

```
    function Build_Fact ( ID    : Identifier;
                          Sons  : List_Of_Facts := No_Facts ) return Clause;
--| where
--|       (Integer_Identifier(ID) or Variable_Identifier(ID)) and
--|            (Length_Of(Sons) > 0) => raise NO_SONS,
--|       (Built_In_Predicate(ID) and then
--|            (Built_In_Arity(ID) /= Length_Of(Sons))) => raise ATTR_ERROR;
        -- Given a predicate symbol ID and a list Sons of facts, this function returns the fact ID(Sons).
        -- Variables and integers cannot have Sons and the built_in_predicates have fixed aritys.

-- Build_Fact forms a complete generator basis for facts.

    function Get_Head ( C : Clause ) return List_Of_Facts;
--| where not Is_Rule(C) => raise ATTR_ERROR;
        -- Given the rule list1 :- list2, this function returns the list list1.

    function Get_Tail ( C : Clause ) return List_Of_Facts;
--| where not Is_Rule(C) => raise ATTR_ERROR;
        -- Given the rule list1 :- list2, this function returns the list list2.

-- The functions Get_Head and Get_Tail together with Is_Rule and the observers on facts form a complete
-- observer basis for Rule (and Clause).

    function Build_Rule ( Head, Tail : List_Of_Facts ) return Clause;
        -- Given lists of facts Head and Tail, this function returns the rule Head :- Tail.

-- Build_Rule together with Build_Fact is a complete generator basis for Rule (and clauses).

-- The following axioms describe the effect the generators have on the observers.

--|axiom
--|  for all C1, C2 : Clause; L1, L2 : List_Of_Facts; ID : Identifier =>

        -- Build_Fact builds facts and Build_Rule builds rules.
--|          not Is_Rule( Build_Fact( ID, L1 )),
--|          Is_Rule( Build_Rule( L1, L2 )),

--|          Get_Identifier( Build_Fact( ID, L1 )) = ID,
--|          Get_Son_List( Build_Fact( ID, L1 )) = L1,
--|
--|          Get_Head( Build_Rule( L1, L2 )) = L1,
--|          Get_Tail( Build_Rule( L1, L2 )) = L2,

        -- Equal(C1, C2) iff all observers return the same values.
--|          Equal(C1, C2) <-> ( Is_Rule(C1) = Is_Rule(C2)) and then
--|                              (( Is_Rule(C1) ->
--|                                 (Get_Head(C1) = Get_Head(C2)) and
--|                                 (Get_Tail(C1) = Get_Tail(C2))) and
--|                                (not Is_Rule(C1) ->
--|                                 (Get_Identifier(C1) = Get_Identifier(C2)) and
--|                                 (Get_Son_List(C1) = Get_Son_List(C2))));
```

```
private

    type Clause_Record;
    type Clause_Rec is access Clause_Record;

end CLAUSE_PACKAGE;

-- Commentary:
-- For a specific implementation, some axioms would be added to reflect the capabilities of the underlying
-- support, such as:

--| axiom
--|     for all C : Clause => Length( Get_Head( C )) = 1;

-- to restrict Rules to be Horn clauses.
```

### 6.4.3 Database Package

```
-- Package Name:
--  Database_package
-- Description:
--  This package defines a database of Clauses.
-- Dependency:
--  List and Clause packages.
-- Author:
--  Neel Madhav

with CLAUSE_PACKAGE, LIST;
use CLAUSE_PACKAGE;

package DATABASE_PACKAGE is

    package DB_PACKAGE is new LIST ( Item => Clause );
    type List_of_Clauses is new DB_PACKAGE.List;

    No_Clauses : constant List_of_Clauses
                 := List_of_Clauses( DB_PACKAGE.Null_List );

end DATABASE_PACKAGE;

-- Commentary:
-- This package captures the meaning of a logic database, and uses only previously defined data structures;
-- very little specification is needed. What might be annotated here are certain natural restrictions on logical
-- databases, e.g. that Integers and Variables may not be asserted as true. This can be accomplished by
-- adding a simple axiom:

--| axiom
--|     for all L : List_Of_Clauses; P : Positive =>
--|         not Constant_Identifier( Get_Identifier( Get_Item( L, P )));
```

### 6.4.4 Substitution Package

```
-- Package Name:
--  Substitution_Package
-- Description:
--  This package defines data-structures which represent substitutions. Substitutions are lists of variable-
-- clause bindings. All free variables in a successful query get bound to clauses as a part of the search for a
-- substitution.
-- Dependency:
--  List, clause and identifier packages.
-- Author:
--  Neel Madhav

with CLAUSE_PACKAGE, IDENTIFIER_PACKAGE, LIST;
use CLAUSE_PACKAGE, IDENTIFIER_PACKAGE;

package SUBSTITUTION_PACKAGE is

    type Binding_Rec is private;
    type Binding is access Binding_Rec;

    -- A Binding is a pair Variable-Clause. The Clause is considered to be bound to the variable.

     BINDING_ERROR : exception;
     -- This exception is raised when an attempt is made to bind a Clause to a non-variable identifier or an
     -- attempt is made to access a Binding that does not exist.

     function Get_Variable ( B : Binding ) return Identifier;
     -- What is the Variable bound by B?

     function Get_Binding ( B : Binding ) return Clause;
     -- What is the Clause bound by B?

     function Set_Binding ( X : Identifier; C : Clause ) return Binding;
--|   where not Variable_Identifier(X) or Is_Rule(C) => raise BINDING_ERROR;
     -- Create a Binding which binds C to X.

     package ANS_PACKAGE is new LIST ( Binding );
     use ANS_PACKAGE;

     type Substitution is new ANS_PACKAGE.List;

     No_Bindings : Substitution;
     -- A list of bindings makes a substitution.

     function Apply ( A : Substitution; C : Clause ) return Clause;
--|   where Is_Rule(C) => raise BINDING_ERROR;
     -- Apply the substitution A to C.

--: function Apply_Binding ( B : Binding; C : Clause ) return Clause;
     -- Apply the binding B to C.
```

```
--: function Apply_Binding ( B : Binding;
--:                           L : List_Of_Facts ) return List_Of_Facts;
    -- Apply for a list of facts.

    function Apply ( A : Substitution;
                     L : List_Of_Facts ) return List_Of_Facts;
    -- Apply for a list of facts.

--|axiom
--|   for all B : Binding;          ID : Identifier; C : Clause;
--|           L : List_Of_Facts;    A : Substitution =>
--|
--|       Get_Variable(Set_Binding(ID,C)) = ID,
--|       Equal( Get_Binding(Set_Binding(ID,C)), C ),

      -- The axioms below describe substitutions.

      -- Applying a single substitution to a Clause.

--|       Equal( Apply_Binding(B,C),
--|           if ( Variable_Identifier(Get_Identifier(C))) then
--|                   if ( Get_Variable(B) = Get_Identifier(C)) then
--|                           Get_Binding(B)
--|                   else    C
--|                   end if
--|           elsif ( Constant_Identifier(Get_Identifier(C))) then
--|                   Build_Fact(Get_Identifier(C),
--|                           Apply_Binding( B, Get_Son_List( C )))
--|           else    C
--|           end if ),

      -- Applying a single substitution to a list of clauses.

--|       Apply_Binding(B,L) =
--|           if CLAUSE_PACKAGE."=" ( L, No_Facts ) then L
--|           else Append( Apply_Binding(B, Head_Of(L)),
--|                        Apply_Binding(B, Tail_Of(L)))
--|           end if,

      -- Applying a list of substitutions to a clause.

--|       Equal( Apply(A,C),
--|           if ( A = No_Bindings ) then C
--|           else Apply( Tail_Of(A), Apply_Binding( Head_Of(A),C))
--|           end if ),

      -- Applying a list of substitutions to a list of clauses.

--|       Apply(A,L) = if CLAUSE_PACKAGE."=" ( L, No_Facts ) then L
--|                    else Append( Apply(A, Head_Of(L)), Apply(A, Tail_Of(L)))
--|                    end if;
```

```
private

    type Binding_Record;
    type Binding_Rec is access Binding_Record;

end SUBSTITUTION_PACKAGE;

-- Commentary:
-- There is a drawback to the recursive specification of this package. Recursion defines an order of sub-
-- stitution, whereas in canonical substitution all variables are assumed to be substituted simultaneously.
-- This is not a problem if further restrictions are added which make order of substitution irrelevant.
-- First, no variable can be substituted twice. Or, in the language of our abstract data types, no Identifier
-- can appear as the variable part of a Binding more than once in a Substitution. One way to annotate this
-- is as a type annotation for Substitution:

    type Substitution is new Ans_Package.List;
--|where S : Substitution =>
--|    for all P1, P2 : Positive =>
--|       P1 /= P2 ->
--|           Get_Variable( Get_Item( S, P1 )) /= Get_Variable( Get_Item( S, P2 ));

-- The second restriction is more difficult: No variable which is to be substituted can occur in the clause
-- part of any other binding in the substitution. This requires another virtual subprogram to capture the
-- concept of "occurring in." For the purpose of brevity, we will restrict our new function to testing the
-- occurrence of a variable in an arbitrary clause:

--: function Occurs_In ( V : Identifier;   C : Clause ) return Boolean;
--| where in Variable_Identifier( V ),
--|     return
--|         if Variable_Identifier( Get_Identifier( C )) then
--|             V = Get_Identifier( C )
--|         elsif Constant_Identifier( Get_Identifier( C )) then
--|           exist P : Positive =>
--|               Occurs_In( V, Get_Item( Get_Son_List( C ), P) )
--|         else -- Integer_Identifier( Get_Identifier( C ))
--|           False
--|         end if;
```

```
-- Now to add the second restriction, we use another type annotation:
--| where S : Substitution =>
--|     for all  P1, P2 : Positive =>
--|         P1 /= P2 ->
--|             not Occurs_In( Get_Variable( Get_Item( S, P1 )),
--|                            Get_Binding( Get_Item( S, P2 )));
```

### 6.4.5 Unification Package

```
-- Package Name:
--  Unification Package
-- Description:
--  This package defines the concept of unification of Clauses.
-- Dependency:
--  Clause, identifier and substitution packages.
-- Author:
--  Neel Madhav


with CLAUSE_PACKAGE, IDENTIFIER_PACKAGE, SUBSTITUTION_PACKAGE;
use CLAUSE_PACKAGE, IDENTIFIER_PACKAGE, SUBSTITUTION_PACKAGE;

package UNIFICATION_PACKAGE is

    UNIFY_ERROR : exception;
    -- This exception is raised if a rule is given an an argument to unify or an attempt is made to get the
    -- unifier of two non-unifiable clauses.

    function Unify ( C1, C2 : Clause ) return Boolean;
--| where Is_Rule( C1 ) or Is_Rule( C2 ) => raise UNIFY_ERROR;
    -- Can C1 and C2 be unified?

    function Unifier ( C1, C2 : Clause ) return Substitution;
--| where not Unify( C1, C2 ) => raise UNIFY_ERROR;
    -- What is the most general unifier for C1 and C2?

    function Unify_List ( L1, L2 : List_of_Facts ) return Substitution;
    -- Returns Unifier of two lists of facts.

--|axiom

-- The general unification algorithm is outlined below.
--|  for all C1, C2    : Clause;
--|           ID1, ID2  : Identifier;
--|           L1, L2    : List_of_Facts =>
```

```
        -- Definition of Unify.
--|     (Get_Identifier(C1) = ID1 and Get_Identifier(C2) = ID2)
--|       -> Unify( C1, C2 )
--|              = if Variable_Identifier(ID1) then True
--|                elsif Variable_Identifier(ID2) then True
--|                elsif Integer_Identifier(ID1) then
--|                    Integer_Identifier(ID2) and ID1 = ID2
--|                elsif Integer_Identifier(ID2) then False
--|                elsif ID1 /= ID2 then False
--|                else
--|                    (Get_Son_List(C1) = L1 and Get_Son_List(C2) = L2)
--|                        -> if Length_of(L1) = Length_of(L2) then
--|                                for all N : 1 .. Length_of(L1) =>
--|                                    Unify( Get_Item( L1, N ), Get_Item( L2, N ))
--|                           else False
--|                           end if
--|                end if,

        -- Definition of Unifier.
--|     (Get_Identifier(C1) = ID1 and Get_Identifier(C2) = ID2)
--|       -> Unifier( C1, C2 )
--|              = if Variable_Identifier(ID1) then
--|                      Append( Set_Binding(ID1, C2), No_Bindings)
--|                elsif Variable_Identifier(ID2) then
--|                      Append( Set_Binding(ID2, C1), No_Bindings)
--|                elsif Integer_Identifier(ID1) then
--|                      No_Bindings
--|                else
--|                      Unify_List( Get_Son_List(C1), Get_Son_List(C2))
--|                end if,
```

```
        -- Definition of Unify_List in terms of Unifier.
--|     Unify_List( L1, L2 )
--|        = if L1 = No_Facts then
--|                    No_Bindings
--|           else
--|                Append( Unifier( Head_Of(L1), Head_Of(L2)),
--|                        Unify_List( Tail_Of(L1), Tail_Of(L2)))
--|           end if;
end UNIFICATION_PACKAGE;

-- Commentary:
-- In the axiom for Unifier, because of the propagation annotation for the Unifier function, we can assume
-- that Unify(C1, C2) is true, which makes the given axiom slightly cleaner.
```

### 6.4.6 Query Package

```
-- Package Name:
--  Query_Package
-- Description:
--  This package defines the concept of querying clauses.
-- Dependency:
--  Clause, Database, Identifier, Substition and Unification packages.
-- Author:
--  Neel Madhav

with CLAUSE_PACKAGE, IDENTIFIER_PACKAGE, SUBSTITUTION_PACKAGE,
      UNIFICATION_PACKAGE, DATABASE_PACKAGE;
use CLAUSE_PACKAGE, IDENTIFIER_PACKAGE, SUBSTITUTION_PACKAGE,
     UNIFICATION_PACKAGE, DATABASE_PACKAGE;

package QUERY_PACKAGE is

   QUERY_ERROR : exception;
   -- This exception is raised if a rule is given as an argument to QUERY or an attempt is made to find
   -- an answer substitution for a failed query.

   function Query ( C : Clause; L : List_of_Clauses ) return Boolean;
--| where Is_Rule(C) => raise QUERY_ERROR;
   -- Does C follow from L?

   function Query_Answer ( C : Clause;
                           L : List_of_Clauses ) return Substitution;
--| where not Query(C,L) => raise QUERY_ERROR;
   -- What are the bindings of variables in C which make the query true?

--:function Query_List ( Q : List_of_Facts;
--:                      L : List_of_Clauses ) return Boolean;
   -- The variable bindings of list Q.
```

```
--:function Query_Answer ( Q : List_of_Facts;
--:                        L : List_of_Clauses ) return Substitution;
--| where not Query_List(Q,L) => raise QUERY_ERROR;
    -- What are the bindings of variables in Q which make the query true?

--:function Sublist ( L1, L2 : List_of_Facts ) return Boolean;
    -- Returns True if L1 is a sublist of L2.

--:function Remove_Sublist ( L1, L2 : List_of_Facts ) return List_of_Facts;
    -- Remove elements of L1 from L2.

--:function Delete_Item( L : List_of_Facts; N : Natural) return List_of_Facts;
--| where in ( N <= Length_of(L)),
--|       return if N = 0 then L
--|              elsif N = 1 then TAIL_OF(L)
--|              else Append( Head_of(L), Delete_Item( Tail_of(L), N-1))
--|              end if;

--|axiom

    -- The constraints on Query are outlined below.

--|    for all C          : Clause;
--|            L          : List_of_Clauses;
--|            N          : Natural;
--|            ID1, ID2   : Identifier;
--|            F1, F2     : List_of_Facts =>

          -- Sublists are defined here.
--|          Sublist( No_Facts, F1 ),
--|          Sublist( Append( C, F1 ), F2 ) =
--|            (exist N : Natural =>
--|                Unify( C, Get_Item( F2, N )) and
--|                Sublist( F1, Delete_Item( F2, N ))),

--|          Remove_Sublist( No_Facts, F1 ) = F1,
--|          Remove_Sublist( Append( C, F1 ), F2 ) =
--|              if Unify( C, Get_Item( F2, 1 )) then
--|                     Remove_Sublist( F1, Tail_of(F2))
--|              else   Remove_Sublist
--|                        (F1, Append( Get_Item( F2, 1 ),
--|                                     Remove_Sublist( Append( C, No_Facts ),
--|                                                     Tail_Of(F2))))
--|              end if,
```

```
--|          Query_List( F1, L ) =
--|              if Is_Equal( F1, No_Facts ) then True
--|              else exist N : Natural =>
--|                       (Sublist( Get_Head( Get_Item( L, N )), F1 ) and
--|                       Query_List( Append( Apply(
--|                           Unify_List( Get_Head( Get_Item( L, N )), F1 ),
--|                           Remove_Sublist( Get_Head( Get_Item( L, N )), F1 )),
--|                                           Apply(
--|                           Unify_List( Get_Head( Get_Item( L, N )), F1 ),
--|                           Get_Tail( Get_Item( L, N )))), L ))
--|              end if,

--|          Query( C, L ) = Query_List( Append( C, No_Facts ), L ),

--|          Query_Answer( F1, L ) =
--|              if Is_Equal( L, No_Clauses ) then
--|                    No_Bindings
--|              elsif exist M : Natural =>
--|                       Sublist( Get_Head( Get_Item( L, N )), F1 ) then
--|                  Append( Unify_List( Get_Head( Get_Item( L, N )), F1 ),
--|                       Query_Answer( Append( Apply(
--|                           Unify_List( Get_Head( Get_Item( L, N )), F1 ),
--|                           Remove_Sublist( Get_Head( Get_Item( L, N )), F1 )),
--|                                                Apply(
--|                           Unify_List( Get_Head( Get_Item( L, N )), F1 ),
--|                           Get_Tail( Get_Item( L, N )))), L ))
--|              else No_Bindings
--|              end if;

end QUERY_PACKAGE;

-- Commentary:
-- In this package we reap the benefits of a hierarchically defined package set. With the help of previous
-- packages and three virtual functions, Prolog-like resolution with substitution can be defined in a single,
-- reasonably compact axiom.
```

# Bibliography


[1] G. Booch. *Software Engineering with Ada.* Benjamin/Cummins, 1987. 2nd Edition.

[2] D. C. Luckham and W. Mann. Methodology for using specification analysis to debug formal specifications. In preparation.

[3] D. C. Luckham, S. Sankar, and S. Takahashi. Two dimensional pinpointing: An application of formal specification to debugging packages. *IEEE Software*, 8(1):74–84, January 1991. (Also Stanford University Technical Report No. CSL–TR–89–379.).

[4] D. C. Luckham and F. W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–23, March 1985.

[5] David C. Luckham. *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs.* Texts and Monographs in Computer Science. Springer-Verlag, October, 1990.

[6] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *ANNA, A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[7] N. Madhav and S. Sankar. Application of formal specification to software maintenance. In *Proceedings of the Conference on Software Maintenance*, pages 230–241. IEEE Computer Society Press, November 1990.

[8] Neel Madhav. An ada-prolog system. In *International Conference on Computing and Information*, pages 340–344, Niagara Falls, Canada, May 1990. (Also Stanford University, Computer Systems Lab technical report CSL-TR-90-437. PAVG technical Report No. 49).

[9] W. Mann. Representation of an Anna subset in predicate logic for specification analysis. Unpublished Technical Report.

[10] Geoffrey O. Mendal. *The Anna-I User's Guide and Installation Manual.* Stanford University, Computer Systems Lab, ERL 456, Stanford, California, release 3C edition, August 1990.

[11] R. Neff. *Ada/Anna Package Specification Analysis.* PhD thesis, Stanford University, December 1989. Also Stanford University Computer Systems Laboratory Technical Report No. CSL–TR–89–406.

[12] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[13] D. S. Rosenblum, S. Sankar, and D. C. Luckham. Concurrent runtime checking of annotated Ada programs. In *Proceedings of the 6th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 10–35. Springer-Verlag — Lecture Notes in Computer Science No. 241, December 1986. (Also Stanford University Computer Systems Laboratory Technical Report No. 86-312).

[14] S. Sankar. *Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs.* PhD thesis, Stanford University, August 1989. Also Stanford University Department of Computer Science Technical Report No. STAN–CS–89–1282, and Computer Systems Laboratory Technical Report No. CSL–TR–89–391.

[15] S. Sankar. A note on the detection of an Ada compiler bug while debugging an Anna program. *ACM SIGPLAN Notices*, 24(6):23–31, 1989.

[16] S. Sankar and M. Mandal. Concurrent runtime monitoring of formally specified programs. Technical Report 90-425, Computer Systems Laboratory, Stanford University, 1990. Also submitted to IEEE Computer.

[17] S. Sankar and D. S. Rosenblum. The complete transformation methodology for sequential runtime checking of an Anna subset. Technical Report 86-301, Computer Systems Laboratory, Stanford University, June 1986. (Program Analysis and Verification Group Report 30).

[18] S. Sankar, D. S. Rosenblum, and R. B. Neff. An implementation of Anna. In *Ada in Use: Proceedings of the Ada International Conference, Paris*, pages 285–296. Cambridge University Press, May 1985.

[19] US Department of Defense, US Government Printing Office. *The Ada Programming Language Reference Manual*, February 1983. ANSI/MIL-STD-1815A-1983.